Memoria del proyecto de Sistemas Informáticos:

Experimentación distribuida con algoritmos genéticos para el aprendizaje de AFs mediante AFPs

Autores: Fernández Carramiñana, Pablo

Jiménez Barral, Carlos

Marchiori, Eugenio Jorge

Profesor director:

Rodríguez Laguna, Ismael

Curso académico:

2007 – 2008

Asignatura:

Sistemas Informáticos,
Facultad de Informática,
Universidad Complutense de Madrid

Autorización

Los autores de esta memoria autorizamos a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a los autores, tanto la propia memoria como el código, la documentación y/o el prototipo desarrollado.

Fdo: Pablo Fernández Carramiñana Fdo: Carlos Jiménez Barral Fdo: Eugenio Jorge Marchiori

Palabras clave para búsqueda bibliográfica

- Autómata finito
- Algoritmo genético
- Autómatas Finitos Probabilistas
- Aplicación distribuida
- Aprendizaje pasivo
- Experimentación

Resumen

Este proyecto incluye el análisis, estudio, desarrollo y pruebas de un algoritmo genético aplicado al aprendizaje pasivo de autómatas finitos (AF). Además, se desarrolló una aplicación distribuida para la realización masiva de pruebas sobre dicho algoritmo.

El aprendizaje pasivo consiste en la inferencia de un AF a partir de una serie de ejemplos, dados al comienzo de la ejecución, de las cadenas que deben ser aceptadas o rechazadas. El algoritmo genético trabaja mejorando en una serie de iteraciones los individuos de la población (conjunto de autómatas finitos probabilistas, o AFPs) hasta que se consigue una solución optima o se deja de mejorar.

En el desarrollo se analizan las características del problema, así como la justificación teórica de las decisiones tomadas en el diseño del algoritmo. Luego se explica la aplicación desarrollada, que consiste en un servidor de problemas, que almacena una serie de problemas y configuraciones posibles para intentar solucionarlo, además de permitir modificar ambas cosas o añadir nuevas y consultar los resultados ya obtenidos. La aplicación también cuenta con un “cliente”, capaz de solucionar problemas de acuerdo a la información facilitada por el servidor y devolver el resultado de la aplicación del algoritmo. Por último, la aplicación dispone de un “administrador”, que es un programa que permite consultar las soluciones, crear nuevos problemas, modificar los existentes, y realizar histogramas con distintos grupos de soluciones filtrados por sus características particulares.

Para poder realizar un estudio más amplio, se desarrollaron distintas formas de afrontar cada una de las partes de algoritmo, así como una implementación muy flexible que permite cambiar cada una de las partes para realizar un gran número de pruebas.

Se crearon dos baterías de pruebas, una inicial para estudiar algunas características básicas y una más pequeña pero homogénea que permitiera realizar un análisis más formal de las características que resultaron de mayor interés en la primera batería. Esto lleva a que el estudio final se realizara sobre más de 70.000 ejecuciones del algoritmo en diferentes problemas y con diferentes parámetros.

Entre los resultados podemos destacar que el algoritmo se comporta correctamente en la mayor parte de las situaciones, y nos permiten aislar ciertas características que llevan en general a un buen comportamiento del algoritmo. También procedemos a comparar el algoritmo con otros ya existentes, lo que nos lleva a ver que el algoritmo resulta aparentemente competitivo en tiempo y a destacar la característica más interesante del algoritmo: que puede encontrar soluciones útiles (que reconocen un gran número de cadenas de ejemplo) incluso cuando no es posible reconocerlas todas con el número de estados solicitado para el autómata solución. Esta característica creemos que es única de este tipo de algoritmo y no la comparten los algoritmos deterministas para solucionar este mismo problema.

Para concluir, analizamos los logros conseguidos en el desarrollo y destacamos las mejores características de nuestro algoritmo y la aplicación desarrolladas. También proponemos opciones para futuros desarrollos, como la aplicación del mismo algoritmo en autómatas de pila (AP).

Summary

This project includes the analysis, study, development and testing of a genetic algorithm used in the passive learning of finite automata (FA). Besides, a distributed application was developed to run a massive number of tests on said algorithm.

Passively learning automata consists on inferring a FA for an example set of accepted and rejected strings, given at the beginning of the execution. The genetic algorithm works by improving the members of a population of probabilistic finite automata (PFA) through a number of iterations, until an optimum solution is reached or the population stops improving from iteration to iteration.

In the development, the characteristics of the problem are analysed, as is the theory behind the decisions taken in the design of the algorithm. After that, we explain the developed application, formed by a problem server, which stores a series of problems and possible configurations to tackle it and at the same time allows for the modification of those settings or the creation of new ones and the retrieval of information from the problems that are already solved. The application also has a "client", capable of solving problems from the information received from the server and sending back the result of applying the algorithm. Lastly, the application has an "administrator", capable of exploring the solutions, creating new problems, modifying existing ones and creating histograms with sets of solutions filtered by their characteristics.

With the idea of making a more profound study, we created different ways to tackle each part of the algorithm, as well as a flexible implementation that allowed to change each part and test a great number of configurations.

We created two testing sets, an initial one to study some basic characteristics and a smaller one (but much more homogeneous one) that allowed us to make a more formal analysis of the characteristics we found more interesting in the first set. Taking this into account, the final study was made from over 70,000 runs of the algorithm with different problems and different parameters.

From the results, we must make emphasis on the facts that the algorithm behaves correctly in most situations and that we are able to establish certain characteristics that lead to a good behaviour of the algorithm. We then compare the algorithm with other that existed before, from what we determine that it seems to be competitive time-wise and to bring forward its the most interesting characteristic: it is able to find useful solutions (those that recognize correctly a big number of the strings in the input set) even when it is impossible to recognize them all given the number of states set for the solutions FA. We believe this characteristic is unique of this kind of algorithm and it certainly isn't shared by the other methods we studied to solve this same problem.

Finally, we study the achievements reached in the development and make emphasis in the best qualities of our algorithm and the developed application. At the same time, we propose options for future developments based on our work, such as the use of the same algorithm to passively learn pushdown automata (PDA).

Índice de contenido

1 Introducción

Nuestro proyecto surgió como una iniciativa orientada a hacer un **experimento relacionado con la teoría de autómatas**. El objetivo era crear una aplicación distribuida que permitiera realizar grandes cálculos relacionados con la obtención de autómatas, máquinas de Turing o lenguajes. El hecho de que fuera una aplicación distribuida permitiría que se pudieran realizar cómputos de gran tamaño. Este experimento podía enfocarse desde distintas vertientes. En particular, planteamos dos posibilidades.

Lo que llamamos el experimento A consistía en un proyecto que involucraba máquinas de Turing. Pero tras un par de semanas, descubrimos que nos resultaba inviable por el gran número de máquinas de Turing que podíamos obtener para pocos estados. Más adelante comentaremos en qué consistía este experimento.

Finalmente, nos decidimos por realizar el segundo experimento, lo que denominamos el experimento B. Viendo que las máquinas de Turing eran demasiado complejas, redujimos el área de estudio a autómatas finitos. De este modo, los lenguajes serían más simples. El nuevo objetivo era obtener un autómata finito de un número determinado de estados y que se ajustara en la mayor medida posible al reconocimiento de unas cadenas de entrada facilitadas por el usuario que debían ser rechazadas o aceptadas. Este problema, el de encontrar dicho autómata, es un **problema NP completo** (en [3]) y por lo tanto una forma habitual de afrontar este tipo de problemas es la "fuerza bruta". Sin embargo, el espacio de búsqueda resultaba muy grande y por ello pensamos en un algoritmo genético, que es más apropiado en estos casos.

Para implementar este algoritmo debíamos tener una población manejable. La decisión de utilizar **Autómatas Finitos Probabilistas (AFPs) en lugar de Autómatas Finitos (AFs)** se debió a que estos últimos presentan inconvenientes en el contexto de los algoritmos genéticos por la mayor dificultad que presenta "cruzarlos" de una forma útil. Los AFP son autómatas cuyas transiciones entre estados se realizan según una probabilidad asignada a dicha transición. De este modo, para un símbolo del alfabeto de entrada puede haber varias transiciones que lleven a diferentes estados destino, cada transición con una probabilidad de producirse. Estos autómatas también nos resultaron interesantes porque **aquellos en los que para cada estado y entrada existe una transición con una probabilidad de ser escogida del 100%, son totalmente equivalentes a AFDs**. Por lo tanto, aunque trabajemos con AFPs dentro del algoritmo, podemos obtener un AFD como resultado. Por lo tanto, uno de los resultados interesantes a estudiar es cómo de cerca está el resultado obtenido por el algoritmo en una ejecución de ser un AF.

El algoritmo genético, pese a ganar incertidumbre en los resultados (dados unos mismos datos de entrada los resultados pueden ser distintos, por sus propiedades estocásticas) respecto a algoritmos deterministas para resolver estos problemas, **puede reducir en gran medida el tiempo de cómputo** necesario para ofrecer resultados y, lo que resulta de gran interés, **puede dar resultados en circunstancias en las que es imposible encontrar una solución mediante métodos deterministas** (esto se analizará en detalle en el desarrollo del proyecto). Para aplicar el algoritmo, necesitamos un conjunto de muestras, AFPs, con diferentes probabilidades entre transiciones, a los que llamaremos miembros de la población o individuos. Sobre ese conjunto de individuos aplicamos operaciones de cruce y mutación, para obtener nuevas muestras de individuos, almacenando sólo los indicados por una función de bondad como los más apropiados para persistir. Sobre los miembros de la población que han "sobrevivido" en cada generación, se vuelven a aplicar las operaciones para generar nuevas muestras y a almacenar las más apropiadas, y así sucesivamente. Finalmente, el algoritmo nos da como resultado un AFP con sus probabilidades

entre transiciones, así como una evaluación de la calidad de la solución, medida en función de la probabilidad de acierto al aceptar o rechazar las palabras de entrada. Además, se devolverá una evaluación del parecido del AFP solución a un AFD, medida en función de la cantidad de transiciones con probabilidad nula o con certeza absoluta.

Pese a que tanto los AFP como el uso de un algoritmo genético mejoran en gran medida los tiempos de cálculo de nuestro estudio, **decidimos utilizar una aplicación distribuida para conseguir un gran número de resultados y poder realizar análisis más rigurosos**. Utilizando una máquina como servidor, éste recibe las peticiones de trabajo y las distribuye entre los ordenadores conectados a él, permitiendo aplicar simultáneamente el algoritmo a distintos problemas pero almacenando las soluciones de forma centralizada, facilitando y reduciendo considerablemente el tiempo necesario para la experimentación masiva.

2 Objetivos

Los objetivos de nuestro proyecto son fundamentalmente dos:

- Analizar, estudiar, desarrollar y probar un **algoritmo genético para la generación de autómatas** a partir de ejemplos (aprendizaje pasivo).
- Crear una aplicación que permita aplicar el algoritmo, estudiar distintas situaciones y principalmente permitir hacer **pruebas masivas en un entorno distribuido.**

Estos objetivos se apoyan en muchas otras partes funcionales de nuestro proyecto que podríamos llamar sub-objetivos, pero que no consideramos objetivos en sí por haber sido creados sólo para alcanzar los dos mencionados antes. Todo el desarrollo de esta memoria y el proyecto en general se basa entorno a éstos.

2.1 Desarrollo de la idea

Desde el comienzo del curso, antes de que se desarrollara la idea en su estado actual, nuestro objetivo fue realizar un proyecto que nos permitiera realizar experimentos relacionados con la Teoría de Autómatas con un enfoque distribuido.

Bastante pronto en el desarrollo del proyecto descartamos una idea inicial de experimento sobre máquinas de Turing debido a que un análisis previo nos llevó a determinar su inviabilidad. Esto se desarrollará con más detalle en el apartado “3. Desarrollo”.

Afortunadamente, teníamos un plan B. Este consistía casi exactamente en realizar lo que es ahora nuestro proyecto, aunque la idea final fue tomando su forma actual una vez que descartamos la idea inicial y a medida que fuimos tomado contacto con la teoría y tecnologías necesarias.

2.2 Método de trabajo

Como se puede ver en los objetivos, nuestro trabajo tiene una componente principalmente teórica y analítica y otra fundamentalmente práctica. Aunque la práctica es principalmente de apoyo a la otra, creemos que por el esfuerzo y la generalidad del método desarrollado, se debe considerar un objetivo por sí mismo.

Para alcanzar el primer objetivo, tratamos de utilizar un enfoque científico, con un **método riguroso para validar nuestras conclusiones** y un esfuerzo de investigación para determinar la relevancia de nuestros esfuerzos.

El segundo objetivo se desarrolló con métodos más cercanos a la **ingeniería del software**. Utilizando conocimientos previos para realizar un diseño coherente que nos permitiera desarrollar una aplicación flexible que se pudiera ir adaptando a las necesidades del proyecto según surgían.

2.3 Resultados esperados

Esperamos cumplir ambos objetivos, y dar muestras suficientes que den constancia de esto. Esperamos también que el método pueda tener implicaciones en la práctica, si no directamente, sí como fundamento para aplicar este enfoque a problemas similares.

También esperamos ser capaces de desarrollar la aplicación de tal forma que sea flexible y útil como base para crear aplicaciones similares para otros experimentos de características parecidas.

2.4 Especificación

Para cumplir el primer objetivo, debemos conseguir lo siguiente:

- Estudiar el "estado del arte" en los temas en los que se apoya el algoritmo, métodos alternativos para conseguir lo siguiente y algoritmos genéticos en general.
- Analizar desde un punto de vista teórico el problema a abordar, considerando las distintas formas de afrontarlo.
- Tomar decisiones de diseño, que concretarán el objetivo en casos de estudio abordables en la práctica y que consideremos de especial interés.
- Justificar de forma teórica las decisiones adoptadas.
- Acotar el ámbito de estudio, para poder alcanzar conclusiones útiles y de interés.

Para el desarrollo de la aplicación y para que esta tenga la utilidad esperada, ésta debe:

- Permitir buscar autómatas finitos a partir de cadenas de entrada.
- Implementar el algoritmo genético en su totalidad, dando la mayor flexibilidad posible para la variación de parámetros y ser modular para cambiar componentes del mismo.
- Ser distribuida, lo que implica la creación de al menos un servidor central y un cliente.
- Funcionar a través de internet, para permitir que clientes en distintas localizaciones puedan colaborar de forma transparente para solucionar problemas.
- Crear un sistema útil y cómodo para añadir problemas y estudiar los resultados almacenados en el servidor.
- Disponer de herramientas para estudios estadísticos, ya que estos serán los de mayor relevancia para validar nuestro proyecto.
- Ser lo más flexible posible, abriendo la posibilidad a extenderla para otros problemas similares.

3 Desarrollo

3.1 Experimento inicial

Nuestra primera intención como proyecto fue la de realizar un experimento relacionado con las clases de complejidad P y NP. La relación entre P y NP es pregunta que lleva muchas décadas sin respuesta. Muchos indicios parecen indicar que posiblemente P es distinto de NP. Sin embargo, dado que no era viable intentar demostrar teóricamente cuál es la relación entre P y NP en sí, consideramos que era un buen objetivo dar algún indicio empírico sobre la desigualdad entre P y NP.

Para dar esos indicios, nos planteamos utilizar Máquinas de Turing Deterministas (MTD) y No Deterministas (MTND). Pretendíamos calcular el número de lenguajes que puede reconocer cada una de ellas para el mismo número de estados. Nuestra intención era calcular cuántos lenguajes eran reconocidos para 2 estados, 3 estados, y posiblemente hasta 4 o 5 estados como mucho, puesto que a priori debe haber muchas MTD según se va aumentando el número de estados. Con estos datos podíamos hacer una gráfica y observar el crecimiento del número de lenguajes en ambas máquinas. Si el número de lenguajes reconocidos por las MTND crecía mucho más rápido el número de los lenguajes reconocidos por las MTD, tendríamos un indicio (aunque no una prueba) de que posiblemente P sería distinto a NP.

Como nuestro objetivo era encontrar todos los lenguajes que generaban las MTD y las MTND, esto suponía que teníamos que calcular todas la MTD y MTND. Pero para poder realizar estos cálculos teníamos que imponer varias limitaciones. Primero, limitar el tamaño de la cinta, puesto que la cinta teóricamente es infinita, pero esto no puede implementarse. También, debería limitarse la longitud de las palabras permitida ya que la cantidad de espacio ocupada podría ser un problema (aunque la aplicación pretendía ser distribuida, si un problema es muy grande tardaría mucho en resolverse y ocuparía mucho espacio al usuario). Y por último, debería limitarse la longitud del número de pasos que pueden darse, puesto que el problema debe finalizar en algún momento.

Necesitábamos guardar los lenguajes que generan las MTD y las MNTD para saber cuántos lenguajes genera cada máquina. Esto implicaba guardar todas las MTD y MTND que cumplieran ciertas características así como sus resultados.

Sin embargo, los problemas de espacio para guardar todas las MT y sobre todo, el tiempo de cómputo que requería una MTND pequeña para reconocer un lenguaje, hizo imposible esta aproximación al problema, puesto que llegamos a la conclusión de que ni siquiera podríamos calcular cuántos lenguajes reconocidos por MTND con 3 estados hay, y disponer sólo del dato de 2 estados sería inútil.

A continuación exponemos en detalle los problemas que encontramos:

Problemas de espacio

Comenzamos planteándonos las necesidades de espacio que tendría el proyecto. Si vemos el problema como un análisis comparativo de MTNDs, podemos pensar que es necesario guardar todas las máquinas posibles. Sabiendo que tenemos más de 2^{200} MTND posibles con 3 estados y que un 1GB son 2^{30} bytes, se ve claramente que necesitaríamos 2^{170} GB para almacenar solo 1 byte por cada máquina. Esto es claramente impracticable.

Un análisis más a fondo del problema nos permitió ver que, en realidad, no sería necesario almacenar todas las MTND, sino sólo aquellas que reconocen un lenguaje reconocido por una MTD, pero con menor tiempo (o número de pasos). Por lo tanto, la cantidad de máquinas a guardar será igual a todas las posibles MTD de 3 estados.

Procedimos a calcular este número y llegamos a la conclusión de que serían tan solo 3888. Y esto es considerando el peor caso en que cada una reconozca un lenguaje distinto, porque en otro caso se pueden volver a agrupar reduciendo aún más el espacio.

Cálculos:

El número de MTD de 3 estados que hay es el siguiente:

$3 * (2^3) * 3^3 = 3^4 * 2^3 = 648$

3 por cada estado

2^3 *por cada transición de estados*

3^3 *por los posibles valores de las transiciones*

Combinaciones de estados finales: $2^3 - 2$ *(no consideraremos cuando todos son estados finales, ni cuando ninguno es estado final)*

$648 * 6 = 3888 = 2^{12}$

Las implicaciones de esto son que en el peor caso de que todos los lenguajes tuvieran todas las posibles combinaciones de letras, bastarían 128 GBytes para almacenar todos los 3888 posibles lenguajes de 16 letras (considerando que almacenamos 4 letras por byte, posible si éstas sólo valen 0, 1 o B).

Cálculos:

Si cada lenguaje puede contener palabras desde 1 hasta n letras, habrá un total de:

$2^n+2^{n-1}+2^{n-2}+... +2^1$ *palabras*

El espacio necesario para almacenar todas las palabras será:

2^{n+1} ** n letras*

Como cada letra puede ser 0,1 o B (blanco), podemos guardar varias letras en un byte. Por ejemplo: 4 letras por byte.

Suponemos que disponemos de un disco duro de 256 Gbytes (2^{38} *bytes)*

*Si cada letra ocupa ¼ byte, se podrán almacenar 2*40 *letras.*

$2^{40} / 2^{n+1} * n = 2^{39-n} / n = 2^{12}$ *(*2^{12} *es el número de MTs que debemos poder guardar)*

$2^{27-n} / n = 1$

$2^{27-n} = n$

$(27-n) \log_2 2 = \log_2 n$

$27-2^m = \log_2 2^m$

$27- 2^m = m$

$27 = m+2^m$

$m = 4$

n = 16

Podemos almacenar palabras de 16 letras.

Creemos que esto bastaría para resolver el problema siempre y cuando para cada MTD almacenemos la MTND que resuelve el mismo lenguaje en el menor tiempo. Ésta es una opción que creemos válida para solucionar los problemas de espacio que surgirían si intentáramos almacenar todas las MTND.

Problemas de ejecución de todas las posibles MTND

Cada MTND se debe ejecutar 2^{n+1} veces (una por cada palabra del lenguaje) donde n es el número de letras máximo de las palabras y fijamos en principio a 16 por las restricciones de espacio fijadas anteriormente.

Para poder realizar la ejecución tenemos que conseguir dos cosas: acotar el tamaño de la cinta (en teoría debería ser infinita) de tal forma que no modifique los resultados y acotar el número de pasos, para evitar la posibilidad de bucles infinitos, limitar el tiempo de ejecución y así eliminar MTNDs que no nos interesan por tener demasiada complejidad.

El tamaño de la cinta podría fijarse arbitrariamente, dado que solo se podría fijar de una forma más segura mediante pruebas. Para darle posibilidades a la MTND de desplazarse, fijamos el tamaño en 4*n (cuatro veces el largo de la palabra, 64 para este caso).

El número de pasos se podría acotar por el que utilizó una MTD en reconocer el mismo lenguaje, ya que si tarda más no nos interesaría. Sin embargo, esto no posible dado que sólo sabemos a qué MTD se corresponde cada MTND (si se corresponde a alguna) cuando ya ha sido reconocido el lenguaje.

Si nos plantemos recorrer el árbol de las MTND en anchura, para poder determinar qué rama llega antes a una solución de manera eficiente en tiempo, podemos estudiar el número de pasos y las implicaciones sobre la memoria necesaria durante la ejecución de una máquina.

Supongamos que acotamos el número de pasos a $n^2.64$, eso son 16384 (2^{14}) pasos. Si para cada estado almacenamos n+2 bytes, necesitaríamos $27^{16384}.(n+2)$ bytes, lo cual es nuevamente imposible (incluso “más imposible” que las otras restricciones por el espacio).

Cálculos:

Definimos el estado de la cinta por las siguientes características:

- *Contenido de la cinta*
- *Posición del lector*
- *Estado de la MT*

Vamos a calcular el espacio necesario para almacenar toda esta información:

Tomamos la decisión de considerar un tamaño de cinta que sea 4 veces el tamaño de la palabra máxima.

*Por tanto, el tamaño que ocupará una cinta será: 4*n*1/4 de byte = n bytes*

Tamaño de la posición del lector:

$log_2 (n*4)$ *bits = 6 bits* $\cong$ *1 byte*

Estado de la MT:

Hay 3 posibles estados ≅ 1 byte

En total, esta información ocupa n+2 bytes. (18 bytes ≅ 2^5 bytes)

Si limitásemos la memoria disponible, con 1GB de memoria (bastante memoria para usar en ejecución) solo podríamos ejecutar 5,25 pasos de la MTND, lo cual no es suficiente. Esto nos lleva a descartar la posibilidad de realizar un recorrido en anchura del árbol.

Cálculos:

A partir de una cinta, en el caso peor podemos obtener para el árbol que desarrolla los pasos de la MTND, hasta 3^3 posibles estados de la cinta siguientes. Por tanto, cada estado de la cinta tendrá 3^3 hijos. Si se pueden hacer 16384 movimientos en la cinta, el árbol tendrá una profundidad de 16384 niveles.

Parece que no podemos permitir tantos movimientos en la cinta. Vamos a calcular cuántos movimientos podemos hacer para poder almacenar todas las cintas. Como cada cinta ocupa 18 bytes, 2^5 bytes, supondremos que cada cliente dispone de 1GB de memoria y que éste cliente será el único que procese esta determinada MT. Dispondremos entonces de 2^{30} bytes.

$2^{30}/2^5 = 2^{25}$ posibles estados de la cinta se pueden almacenar

Calculamos cuántos niveles del árbol representan estos 2^{25} posibles estados. Como en cada nivel del árbol el factor de ramificación es 3^3,(27), tenemos:

$27^x = 2^{25}$

$x = log_{27}\ 2^{25}$

$(25\ log_2 2) / log_2\ 27 \cong 5{,}25$ pasos permitidos en la cinta.

Éste número de pasos es tan bajo que nos impide hacer una ejecución normal de una MT.

¿Qué sucedería si realizáramos un recorrido en profundidad?

- Como ya indicamos, es necesario limitar el número de pasos. Si lo fijamos a 16384 como antes, solo necesitaríamos aproximadamente 1 MB de memoria, lo que es razonable e incluso permitiría relajar alguna otra restricción.

Sin embargo, el mayor problema en la ejecución no es de espacio, sino de tiempo. Mientras pensábamos en las restricciones de espacio, nos dimos cuenta de que tenemos un importante problema de tiempo.

Incluso haciendo el análisis en el mejor caso, en que todas las máquinas encuentren la solución en la primera rama que recorren y que pudieran realizar 2^{10} pasos por segundo, dado que cada máquina se debe ejecutar una vez por cada palabra y hay 2^{200} MTNDs, necesitaríamos:

$2^{200} * 2^{16+1} * 2^{14} / 2^{10}$ (MTNDs * Posibles palabras * Prof recorrida / Pasos por segundo) =

$= 2^{221}$ segundos.

Esta cantidad de tiempo es inalcanzable. De hecho, nos hizo pensar que incluso si pudiéramos estudiar una MTND por segundo, necesitaríamos 2^{200} segundos. Esto implica que si repartiéramos el trabajo entre toda la humanidad (suponiendo que cada persona tiene un ordenador, y que hay aproximadamente 2^{33} personas), e incluso si pudiéramos procesar 2^{10} MTNDs por segundo (tarea seguro imposible) necesitaríamos 2^{157} segundos, o 2^{144} años, o 2^{120} millones de años,

aproximadamente 2^{85} veces la vida del universo.

Por estas razones creemos que es absolutamente imposible estudiar el problema planteado mediante fuerza bruta o, de hecho, mediante cualquier otro método algorítmico o heurístico.

3.2 Definiciones y justificaciones teóricas

A continuación pasamos a definir distintas cuestiones que utilizaremos a lo largo del desarrollo del proyecto y a justificar teóricamente algunas de las decisiones que tomamos a priori.

3.2.1 Autómata Finito Determinista

Definimos un autómata finito como una 5-tupla de la forma (Σ, S, s_0, δ, F) donde:

- Σ es un conjunto finito, que representa el alfabeto de entrada
- S es un conjunto finito y no vacío de estados
- s_0 es el estado inicial, debe ser un elemento de S
- δ es la función de transición, definida como $\delta : S \times (\Sigma \cup \{\varepsilon\}) \rightarrow S$
- F es el conjunto de estados finales

Ésta es la definición que utilizaremos a lo largo del proyecto. Sin embargo, en algunos casos nos referiremos a los autómatas finitos como autómatas finitos deterministas, dado que tienen las mismas capacidades expresivas.

3.2.2 Autómata Finito No Determinista

La definición de un AFND es equivalente a la de AFD, pero con una distinción en la función de transición, que se define como:

$$\delta : S \times \Sigma \rightarrow P(S)$$

Donde P(S) es el conjunto de partes de S. Además, aquí se debe hacer otra distinción, dado que en nuestro caso al referirnos a AFND consideraremos la transición vacía ε por lo que queda:

$$\delta : S \times (\Sigma \cup \{\varepsilon\}) \rightarrow P(S)$$

dado que representa a los mismos autómatas y la conversión entre uno y otro es automática.

La capacidad expresiva de los AFD y AFND es equivalente, y además existe una forma algorítmica de pasar de los unos a los otros, por lo que en general no se hará distinción entre ellos. Cuando nos referimos a cualquiera de éstos, se pude entender que estamos hablando del conjunto de AF y todas las conclusiones alcanzadas para unos son equivalentes para los otros.

3.2.3 Autómatas Finitos Probabilistas

Formalmente definimos los Autómatas Finitos Probabilistas (AFP) como una 5-tupla, de la misma forma que los AF y AFD, pero:

$$\delta = S \times (\Sigma \cup \{\varepsilon\}) \rightarrow P(S \times p)$$

F es un conjunto de $(S \times p)$

Donde p es la probabilidad de que se coja la transición en la función de transición o la de que se acepte una cadena cuando llega, representada como un número real entre 0 y 1. La suma de todas estas probabilidades para las transiciones que parten de un estado debe ser 1, formalmente:

$$\forall\, o \in S / \left(1 = \sum p\,,\, \delta(o,c) = (s\,,\, p) \wedge \forall\, s \in S \cup \{\varepsilon\} \wedge \forall\, c \in \{0,1\}\right)$$

Una de las más importantes diferencias es que el resultado de procesar una cadena con este tipo de autómatas no es un valor booleano de aceptada o rechazada como lo es en los otros casos sino que es un valor real entre 0 y 1.

Cabe notar aquí que éste es el tipo de autómatas que utilizará nuestro algoritmo genético.

Otra característica interesante (que nos será de gran utilidad en el proyecto) es que si las probabilidades son todas 0 y 1, el AFP será equivalente a un AF.

3.2.4 Validez de los resultados para cadenas con más símbolos

En el desarrollo del proyecto, utilizaremos como conjunto de símbolos para las cadenas el 0 y el 1. Sin embargo, hay que tener en cuenta que los resultados alcanzados tienen validez para cadenas con más símbolos dado que se puede demostrar teóricamente la equivalencia entre autómatas de 2 símbolos con aquellos de 3 o más.

No procederemos a realizar una demostración formal dado que no lo consideramos necesario, pero el principio detrás de la demostración es el de reemplazar todos los símbolos del lenguaje con más de dos por 0 y 1. O sea, lo mismo que hacen los ordenadores continuamente.

3.2.5 El algoritmo genético

El objetivo del proyecto era obtener una aplicación que encontrara autómatas finitos de N estados (indicado por el usuario) que aceptara las palabras solicitadas por el usuario y rechazara otras también proporcionadas por él, mediante un algoritmo genético. Obtener un autómata por fuerza bruta parecía poco eficiente (en *4.3.2.3.Comparación* se estudia en detalle), los algoritmos óptimos no son mucho más eficientes y tienen otros inconvenientes (estudiados en la misma sección) y por ello se consideró que un algoritmo genético podía ser el camino más adecuado para hacerlo.

El proceso del algoritmo genético se basa en una población de individuos con unas características propias (en nuestro caso autómatas finitos probabilistas, cuya característica está formada por los valores de probabilidades que tienen para las transiciones de cada estado del autómata a cualquier otro estado). Esta población irá evolucionando, al igual que sucede con las especies de la naturaleza, cruzando individuos de su población. Al cruzar miembros de la población, surgirán nuevos miembros que tendrán características del padre y de la madre. En nuestro caso, los autómatas que surgirán a partir de otros dos tendrán unas probabilidades parecidas a las de sus padres. Además de esto, hay una probabilidad de que los nuevos miembros muten, variando más las características de este miembro.

Este proceso de coger dos miembros y cruzarlos se realiza un gran número de veces con muchos ejemplares de toda la población (el conjunto de individuos que es un parámetro del algoritmo). De todos los miembros obtenidos, se clasifican y se escogen los mejores (éstos pueden ser padres o hijos) mediante una función que calcula la bondad de dicho miembro (ese autómata probabilista). Se repite el proceso a partir de los escogidos un número de veces, hasta que se escoge el mejor autómata obtenido en el proceso.

A continuación pasamos a describir cada uno de los elementos de un algoritmo genético (algunos son generales y otros particulares de los que implementaremos nosotros)

Cruzadores

Los cruzadores son los elementos que se encargan de cruzar dos individuos de la población para obtener nuevos miembros. Existen diferentes formas de realizar estos cruces. La idea general consiste en coger características de ambos y mezclarlas en un individuo nuevo. Este elemento es una de las claves del algoritmo genético, ya que se encarga del proceso de ir cruzando miembros para ir obteniendo nuevas poblaciones mejores que las anteriores.

Puede haber características de individuos que tengan un valor en un dominio de gran tamaño. Por ejemplo, en nuestro caso, cada transición tiene una probabilidad asignada. Esta probabilidad tiene un valor real entre 0 y 1. El individuo resultante del cruce de sus padres, puede tener un valor de esta característica igual al de uno de sus padres. Pero hay más alternativas, como coger un valor que esté entre los dos valores de sus padres, por ejemplo.

Mutadores

El mutador es la parte del algoritmo genético que se encarga de generar un individuo que no sólo es fruto de un cruce de sus padres, sino que también sufre una mutación, que consiste en una variación de sus características. Esta parte de un algoritmo genético es muy importante, puesto que con los cruces, los individuos resultantes siempre se parecen a sus padres.

Es posible que el individuo que representa la solución tenga algunas características muy distintas a las de los individuos de la población que se van generando cada iteración. Por ello, es posible que, como los individuos que se van generando siempre se parecen bastante a sus padres, no consigan mejorar lo suficiente para llegar a la solución. De aquí surge la importancia del mutador. Cuando se crea un individuo nuevo, hay una probabilidad de que el mutador cambie algunas características, sin tener en cuenta si estas nuevas características se parece a alguna de los padres.

Cuando se están estancando los individuos de una población en unas características concretas, es el mutador el que posiblemente consiga obtener algún individuo nuevo y mejor, y permita que la población siga evolucionando en las siguientes iteraciones.

Calculadores de Bondad o Función de bondad

Además de cómo se cruzan y se mutan los individuos de la población en un algoritmo genético, es muy importante escoger una forma adecuada de seleccionar qué miembros de toda la población son los mejores y van a ser seleccionados para pasar a la siguiente generación.

Este proceso de selección depende mucho del problema concreto que se esté tratando. En nuestro caso, para calcular qué autómatas son mejores, debemos evaluar con qué probabilidad aceptan las palabras que el usuario ha requerido que se acepten y con qué probabilidad se rechazan las palabras que el usuario ha requerido que se rechacen. Pero esta forma de evaluarlo se puede llevar a cabo de distintas formas, valorando más algunos aciertos que otros, o considerando otros factores como por ejemplo, considerar mejor un autómata que se acerca más a ser determinista que otro que no se acerca.

Poblaciones

El tamaño de la población es un factor importante en un algoritmo genético. La población es el número de individuos máximo que se obtiene al cruzar individuos durante una generación. Cuando se alcanza el límite de población, ésta no puede seguir creciendo, y se termina esa generación.

Cuanto mayor es la población, se manejan más individuos, y por tanto, la probabilidad de que se obtenga una buena solución aumenta. Sin embargo, a la hora de implementar un algoritmo genético, hay que tener en cuenta el compromiso entre población máxima y eficiencia, ya que si se aumenta mucho el tamaño de población máxima, se crean más autómatas por generación, y esto repercute en el tiempo de ejecución del algoritmo, además de que aumenta la cantidad de memoria necesaria para ejecutar el algoritmo.

Muestras

Otro factor importante es el número de individuos que pasan de generación. Al terminar una iteración, se eligen los mejores individuos, que serán la población inicial de la siguiente generación, y los que se cruzarán para obtener nuevos individuos. Llamaremos a esta cantidad de individuos: muestras. En este caso, el compromiso es otro.

Por un lado, está la ventaja de que cuantos más individuos haya en la población inicial, más variedad habrá en la población final de esa generación. Sin embargo, hay un inconveniente. Si este número de individuos escogidos es alto, ocurrirán dos cosas.

Las posibilidades de cruce entre ellos disminuirán, y es posible que dos individuos potencialmente buenos, tengan la mala suerte de que se crucen una sola vez resultando en un individuo de calidad pobre.

Y el segundo inconveniente de tener un número de individuos alto es que entre estos individuos puede haber individuos muy buenos, pero, debido a que el número escogido es alto, habrá individuos que no serán tan buenos, y se perderá tiempo cruzándolos porque posiblemente resulten en individuos que tendrán poco valor.

3.2.6 Valoración de resultados

Para todos los resultados calcularemos dos variables, que nos permitirán saber qué tan bien se ajusta el AFP obtenido a resolver el problema o reconocer el lenguaje. Una de estas características la llamamos “Bondad de reconocimiento” o simplemente “Reconocimiento” e indica la probabilidad de que una cadena sea aceptada o rechazada según corresponda. La otra característica la llamamos “Parecido a AF” e indica que tanto se parece el AFP obtenido a un AFD. Esta segunda característica es interesante porque nuestro objetivo último es obtener AFDs y los AFPs los estamos utilizando de forma instrumental.

Reconocimiento

Es la media de las probabilidades de aceptar una cadena aceptada por el lenguaje y las probabilidades de rechazar una cadena rechazada por el lenguaje. Matemáticamente:

$$R = \frac{\sum p(s) / s \in Aceptadas + \sum (1 - p(s)) / s \in Rechazadas}{|Aceptadas| + |Rechazadas|}$$

donde:

- p(s) es la probabilidad de una cadena s de ser aceptada
- Aceptadas es el conjunto de las cadenas aceptadas
- Rechazadas es el conjunto de las cadenas rechazadas

Parecido AF

Esta variable es la media de los valores de todas aquellas transiciones que tienen una probabilidad mayor a 0,5 de ser elegidas. Matemáticamente, esto se define como:

$$P = \frac{\sum p / \delta(o, c) = (f, p) \wedge o \in S \wedge f \in S \cup \{\varepsilon\} \wedge c \in \{0,1\} \wedge p > 0{,}5}{|S| * 2}$$

donde:

- S es el conjunto de estados
- δ es la función de transición del AFP

3.3 Decisiones de diseño

3.3.1 Problema de los AF como individuos de la población

Si el objetivo de nuestra aplicación es obtener autómatas finitos deterministas, ¿por qué elegimos utilizar AFPs? Al utilizar un algoritmo genético para hallar el autómata que buscábamos, tendríamos que tener poblaciones de autómatas que tendríamos que cruzar. En cada iteración del algoritmo, se mezclan aleatoriamente entre ellos, y al final de cada paso se evalúa la calidad o bondad de todos los autómatas para quedarnos con los mejores. A la hora de cruzarlos o mezclarlos, sus transiciones se entremezclan, cogiendo unas de un autómata y otras de otro. Aunque estos dos autómatas fueran buenos, es muy posible que al entremezclar sus transiciones la bondad del autómata resultante empeore mucho, ya que el cambio de una transición en un autómata determina el lenguaje entero. Por este motivo, si utilizábamos autómatas finitos como individuos de las poblaciones que íbamos a utilizar en el algoritmo genético, la bondad de los autómatas iría a saltos, pudiendo tardar mucho en encontrar una solución.

3.3.2 Ventaja de los AFPs como individuos de la población

Encontramos que los AFPs iban a suponer una importante ventaja para el algoritmo. Al tener transiciones con unas probabilidades asignadas, podíamos conseguir que el proceso de cruce entre autómatas fuera más sutil que si se tratasen de autómatas finitos. En este caso, al cruzar autómatas podíamos alterar algunas probabilidades ligeramente, aumentando o reduciendo la bondad del autómata, pero en cualquier caso, avanzando de forma más pausada en la obtención del autómata final.

3.3.3 Representación en memoria de los AFPs

Existen numerosas formas de representar los AFPs en memoria, y cada una de ellas tiene ventajas y desventajas. Nosotros utilizamos al menos dos representaciones distintas, una que presenta ventajas en espacio y para calcular la probabilidad de que una palabra sea aceptada (representación matricial) y otra que es más sencilla de dibujar en pantalla y de modificar (añadiendo estados, transiciones, etc.).

Respecto a la representación matricial, que creemos que es de mayor interés dado que es la utilizada por el algoritmo en sí, consiste en una matriz tridimensional de float de tamaño 2 x n x (n+1). En cada posición A x B x C se representa la probabilidad de la transición que va de B a C con entrada A. El tamaño lleva un n+1 porque hay una transición extra a un estado trampa que equivale a que la transición no exista, pero que es necesario para que la suma de todas las transiciones que parten de un estado sea 1. En esta representación hay otros detalles, como un vector que representa la probabilidad de ser final de cada estado.

3.4 Proceso de desarrollo

Primer prototipo

El primer prototipo que realizamos consistía en una aplicación que recibía las palabras de entrada mediante un fichero de texto. En ese fichero se incluían las palabras que debían ser aceptadas, las palabras que debían ser rechazadas, el número de estados que debía tener el autómata, y algunas opciones parametrizables del problema. Se devolvía el resultado por consola, y éste consistía en una lista de todas las transiciones de cada estado del autómata a cualquier otro estado, y la probabilidad de que existiera la transición con 0 o con 1. Se incluía también un valor de bondad que indicaba la precisión del autómata, es decir, la probabilidad de aceptar o rechazar correctamente las cadenas que se incluían en el archivo de entrada según correspondiera.

Este prototipo resolvía el problema mediante el algoritmo genético. Sin embargo, al ser un prototipo sólo disponía inicialmente de un tipo de cruzador, mutador y calculador. La población y muestras sí se podían cambiar, pero se debía hacer modificando un valor en el código fuente. Además, este prototipo todavía no tenía la capacidad de trabajar de forma distribuida.

El uso de este primer prototipo nos permitió hacer las primeras pruebas para comprobar el buen funcionamiento del algoritmo genético y de los AFP. Sin embargo, pronto quedó obsoleto para su uso en cuestiones más complejas.

Desarrollo de la interfaz

Conforme trabajábamos sobre el primer prototipo, se hizo patente la necesidad de crear una interfaz gráfica para la herramienta debido tanto a que el sistema final debía disponer de ella, como a nuestra propia comodidad a la hora de realizar pruebas.

Comenzamos creando una interfaz que permitía, por un lado generar un AFD de forma aleatoria, indicando el número de estados deseado para éste, y por otro, dibujar un AFD sobre un panel, pudiendo configurar todas sus opciones. Para estas dos opciones, añadimos la funcionalidad de generar aleatoriamente cadenas aceptadas y rechazadas por el autómata creado.

De este modo, ya podíamos hacer más pruebas con autómatas más grandes y generar palabras aceptadas y rechazadas más cómodamente.

Servidor y la base de datos

Para poder dar forma a la parte distribuida de la aplicación, era necesario disponer de un servidor que gestionara los problemas y los distribuyera entre las distintas máquinas cliente, así como una base de datos que permitiera almacenar dichos problemas y las soluciones a los mismos. Para crear este servidor y hacer las implementaciones necesarias fue necesario investigar en distintas áreas.

Investigaciones previas sobre aplicaciones distribuidas

Debido a que nuestra intención era desarrollar una aplicación distribuida semejante a la usada por el programa “Seti@home”, decidimos buscar información sobre éste. Descubrimos que el programa estaba construido sobre un sistema llamado BOINC. Dicho sistema, es un proyecto no comercial de la universidad de Berkeley, que da soporte a diferentes aplicaciones distribuidas para investigaciones que requieren una gran potencia de cálculo. BOINC distribuye los diferentes

problemas de la aplicación construida sobre él, entre los distintos ordenadores alrededor del mundo, que tienen instalado el programa cliente.

Pese a que aparentemente BOINC cubría nuestras necesidades para la implementación de la aplicación distribuida, eran necesarios unos requisitos formales que no podíamos cumplir, así que tuvimos que desechar esta opción.

Otra opción que planteamos fue el Globus Toolkit. Se trataba de un software libre que permitía desarrollar aplicaciones distribuidas, que era nuestro objetivo. Sin embargo, la gran complejidad del proyecto y las numerosos funcionalidades de las que no íbamos a sacar provecho, así como la curva de aprendizaje para el uso de esta tecnología (no apropiada para el desarrollo de un proyecto de fin de carrera) nos llevaron a rechazar también esta tecnología.

Implementación final del servidor

Puesto que las opción del BOINC no era viable y la del Globus Toolkit no era favorable, tuvimos que buscar otras opciones. Teniendo algunos conocimientos previos en el desarrollo de WebServices y en el uso de servidores de aplicaciones (en particular JBoss), decidimos aprovechar los conocimientos e ideas adquiridos investigando otras tecnologías para desarrollar nuestro propio sistema desde cero. Podría parecer *a priori* que esto puede ser más costoso, pero como nos limitamos a implementar sólo lo estrictamente necesario nos permitió ganar tiempo frente a las otras alternativas y crear una aplicación que cumpliera todos nuestros objetivos.

Decidimos ubicar el servidor en el ordenador de uno de los miembros del equipo, por cuestiones prácticas, aunque es muy sencillo pasarlo a otros sistemas tanto con Windows como con un sistema Unix. Utilizamos una maquina virtual con Virtual PC de Microsoft por cuestiones de seguridad y para simplificar más la tarea de trasladar el servidor si se diera algún problema, y creamos un DNS dinámico, para poder adaptarnos a los cambios de IP del servidor debidos a no disponer de un ISP con IP estática.

Para la implementación de la base de datos hemos usado MySQL, dónde almacenamos todos los datos relativos al problema y sus configuraciones, así como las soluciones que se den, cuando el problema se resuelva.

Finalmente, el sistema funciona correctamente y permite la colaboración distribuida a través de Internet para la solución de los problemas que nos interesan en este proyecto.

Creación de las dos aplicaciones

Para poder distribuir entre la mayor cantidad posible de personas nuestra aplicación, y que estas pudieran colaborar resolviendo problemas pero sin afectar la integridad de la base de datos, creamos dos aplicaciones.

Una para los administradores, que dispone de herramientas para modificar la base de datos añadiendo problemas nuevos, con sus palabras aceptadas, configuraciones y demás información necesaria, ver estadísticas de la soluciones y resolver problemas particulares sin necesidad de subirlos a la base de datos.

La otra aplicación, la de los clientes que prestarían su capacidad de cómputo, tiene una funcionalidad más limitada, que sólo permite buscar problemas de la base de datos para darle solución y ver las respuestas generadas para aquellos clientes que tenga curiosidad de ver el trabajo que está realizando su sistema.

Optimizaciones y ampliaciones

Con el servidor, la base de datos y la interfaz operativas, pasamos a la fase de optimización del algoritmo genético. Haciendo uso de unas primeras pruebas, comprobamos la calidad de los mutadores, cruzadores, calculadores de bondad y resolutores, y llevamos a cabo la implementación de otros para intentar mejorar los anteriores y ofrecer más opciones de configuración.

Todos los elementos se beneficiaron de pequeñas mejoras en su funcionamiento y comprobamos, que según el tipo de problema, era más óptimo el uso de un mutador, cruzador o calculador distinto. Sin embargo, en el caso de los resolutores no fue así. Se realizaron dos implementaciones, pero la única diferencia entre ellas es de eficiencia mientras que los resultados son los mismos (estos se detalla en *3.5.4."Resolutores"*).

Fase de pruebas

Una vez tuvimos todas las partes de la herramienta completadas y optimizadas, pasamos a la fase de pruebas. Introdujimos una batería de pruebas, con diferentes problemas y configuraciones para éstos, en la base de datos.

Estás pruebas intentaban ver los límites de la herramienta, así como las mejores configuraciones para el algoritmo genético y sobre todo, las virtudes y defectos de los AFP combinados con el algoritmo genético para la resolución de este tipo de problemas.

Por medio del cliente dejamos la herramienta funcionando resolviendo los problemas propuestos en diferentes máquinas.

3.5 El algoritmo genético

Al utilizar un algoritmo genético para la búsqueda de AFD, vimos la posibilidad de implementar los distintos elementos (cruzadores, mutadores y función de bondad) de distintas maneras. Para poder determinar qué era conveniente, los creamos de tal forma que fueran configurables y reemplazables (mediante interfaces y factorías) con el fin de poder identificar las mejores implementaciones mediante pruebas. A continuación se describen las implementaciones de dichos elementos que incluimos en proyecto, pero las valoraciones prácticas de cada uno se dejan para el análisis de resultados más adelante.

3.5.1 Cruzadores

Primer cruzador

El primer cruzador implementado escoge dos miembros de la población aleatoriamente y los cruza de la siguiente forma:

- Calcula un valor real aleatorio entre 0 y 1 para cada estado del primer autómata (lo llamaremos p(E)).
- Multiplica por p(E) el valor de las transiciones del primer autómata que tienen como origen E.
- Suma la probabilidad del segundo autómata para las transiciones de E, multiplicando antes los valores por 1 – p(E).

De esta forma, se calculan nuevas probabilidades para las transiciones de todos los estados del nuevo autómata y se garantiza que la suma de todas las que parten de cada estado para una entrada dada sigue siendo 1 (restricción puesta a los AFP por definición).

Este cruzador incluye un factor aleatorio, que puede ayudar al algoritmo a evitar estancamientos, pero al mismo tiempo corre el riesgo de dejar los valores estancados dentro de cierto rango impuesto por los autómatas originales por la forma en la que calcula los nuevo valores.

Segundo cruzador

El segundo cruzador implementado escoge los miembros de la misma forma, aleatoriamente. Luego, para cada estado E:

- Genera un booleano aleatorio B.
- Si B es cierto, copia las transiciones desde E como son en el primer autómata.
- Si B es falso, copia las transiciones desde E como son en el segundo autómata.

En este caso también el AFP creado mantiene las restricciones de la definición. Este autómata puede dar buenos resultados si el orden de los estados implica igualdad estructural (esto es, que las cadenas que se reconozcan o se deben reconocer de dos estados en la misma posición en diferentes autómatas sean las mismas), sin embargo esto no lo podemos asegurar por lo que los resultados conseguidos con este cruzador pueden llegar a variar mucho. Sería interesante intentar determinar una "equivalencia estructural" entre estados de los autómatas, pero esto es particularmente complicado para los AFP e incluso puede ser imposible en la mayoría de los casos.

Tercer cruzador

El tercer cruzador implementado, en lugar de escoger los miembros aleatoriamente, se encarga de cruzar cada uno de los miembros de la población con todos los demás, tantas veces como sea necesario para conseguir la población objetivo. El proceso de cruce, sin embargo, es el mismo que en el primer cruzador.

Esperamos que este cruzador se comporte de manera similar al primero para poblaciones grandes, pero puede resultar ventajoso para poblaciones pequeñas porque nos aseguramos de una distribución más homogénea de las características de los padres.

Cuarto cruzador

Este cruzador es una mezcla del tercero y el primero. Escoge algunos de los individuos a cruzar ordenadamente, y el resto aleatoriamente. El proceso de cruce es igual al del primer cruzador.

3.5.2 Mutadores

Primer mutador

El primer mutador actúa sobre un AFP recorriendo todas las transiciones que salen de cada estado, en cada una hace una de las siguientes cosas:

- Multiplica todas las probabilidades de las transiciones que salen de ese estado por sí mismas, y luego ajusta esos valores para que la suma de las probabilidades sigan sumando 1.
- Se asigna para todas las transiciones que salen de un estado el valor 0, excepto a una, que se le asigna el valor 1. De este modo, se aproxima el AFP a un AFD, ya que para esa transición, su destino es otro estado con probabilidad 1.
- No se hace nada con esa transición.

Segundo mutador

El segundo mutador recorre todos los estados, y con una probabilidad del 0,05 realiza la siguiente operación en cada uno:

- Busca el mínimo valor (distinto de 0) de todas las probabilidades de las transiciones que salen de un estado.
- Resta este valor a todas las probabilidades y luego reajusta las probabilidades para que sigan sumando 1.

Al hacer esto, las probabilidades se alejan de valores intermedios. Las probabilidades cercanas a 0, se aproximan más a 0, y las cercanas a 1, se aproximan a 1, con el fin de que el AFP se parezca un poco más a un AFD.

Tercer mutador

El tercer mutador utiliza un sistema muy similar al primero. Sin embargo, en el primer mutador, para una transición podían, o modificarse todas sus probabilidades multiplicándose, o se podía transformar una transición en una con todo probabilidades 0, y una con probabilidad 1, o no ocurrir

nada. En este mutador, pueden ocurrir los dos primeros casos simultáneamente para una transición. Además, la probabilidad de que suceda el segundo caso se aumenta.

3.5.3 Calculadores de bondad

Calculador de bondad simple

Este calculador de bondad, simplemente realiza la media entre las probabilidades de ser aceptadas o rechazadas las palabras de entrada según corresponda.

Calculador de bondad cuadrático

Este calculador de bondad es semejante al simple. Sin embargo, éste calcula los cuadrados de las probabilidades antes de hacer la media. De este modo, si una palabra tiene una probabilidad baja de ser reconocida correctamente (ya sea aceptada o rechazada), al elevarla al cuadrado se reduce relativamente su valor aún más. Sin embargo, cuanto más cercana a 1 es la probabilidad, menos se reduce el valor al elevarla al cuadrado. De este modo se favorecen los autómatas que reconocen moderadamente bien todas las palabras frente a los que reconocen muy bien algunas pero muy mal otras.

Calculador de bondad balanceado

Este calculador de bondad da la misma importancia a las palabras aceptadas que a las rechazadas. En la aplicación, el número de palabras aceptadas y rechazadas que se den como entrada puede fácilmente ser muy distinto (hay lenguajes que queremos que acepte muy pocas palabras, y queremos que rechace todas las demás). Utilizando cualquiera de los anteriores calculadores en un problema de estas características (por ejemplo, pocas palabras aceptadas y muchas rechazadas) simplemente calcularía la media entre todas las probabilidades, y por ejemplo un autómata que tuviera todos los estados como estados de rechazo, tendría un alto valor de bondad, pero esto no nos interesa.

Este calculador tiene en cuenta que como el número puede ser muy distinto, considera igual de importante acertar en el reconocimiento de todas las palabras aceptadas y todas las palabras rechazadas. De este modo, este calculador evita estancamientos en problemas con gran diferencia entre el número de palabras de entrada aceptadas y rechazadas, y particularmente tiende a dar valores mucho peores a aquellos autómatas que rechazan todas las palabras aceptadas o aceptan todas las rechazadas cuando dichos conjuntos son menores.

Calculador de bondad de preferencia determinista

Este calculador tiene en cuenta más factores a la hora de evaluar si un autómata es bueno o no. La idea de este calculador es escoger antes un autómata que se parece más a un autómata finito determinista que otro que no se parece aunque tenga un porcentaje de acierto en las palabras de aceptación y rechazo parecido (esto se calcula de igual manera que en el calculador de bondad balanceado). Para ello, este algoritmo busca entre todas las transiciones que salen de cada estado. Como todas las probabilidades de las transiciones que salen de cada estado deben sumar 1, busca estados que tengan probabilidades mayores de 0.5, y los evalúa como algo positivo. Si las encuentra, entonces es que el resto de transiciones que salen de ese estado son menores de 0.5 y

podemos considerar la transición más importante que sale de ese estado la que tiene el valor mayor de 0.5. Si no encuentra ninguna transición mayor de 0.5 es que el parecido con un autómata finito determinista es más lejano, ya que puede ser que las probabilidades estén muy repartidas (por ejemplo puede haber transiciones a 4 estados distintos con probabilidades: 0.4, 0.3, 0.2, 0.1).

Comparación entre los calculadores de bondad

De esta comparación se excluye el calculador de bondad de preferencia determinista, dado que utiliza más factores que la probabilidad de cada palabra de ser aceptada o rechazada. Sin embargo, una de las componentes más importantes de este calculador será igual al valor del calculador balanceado que si se estudia a continuación.

Ejemplo 1

Aceptadas	**Rechazadas**
0,7	0,8
0,6	0,9
0,8	
0,6	

Ejemplo 2

Aceptadas	**Rechazadas**
1	0,7
1	0,8
0,2	0,6
0,4	0,9

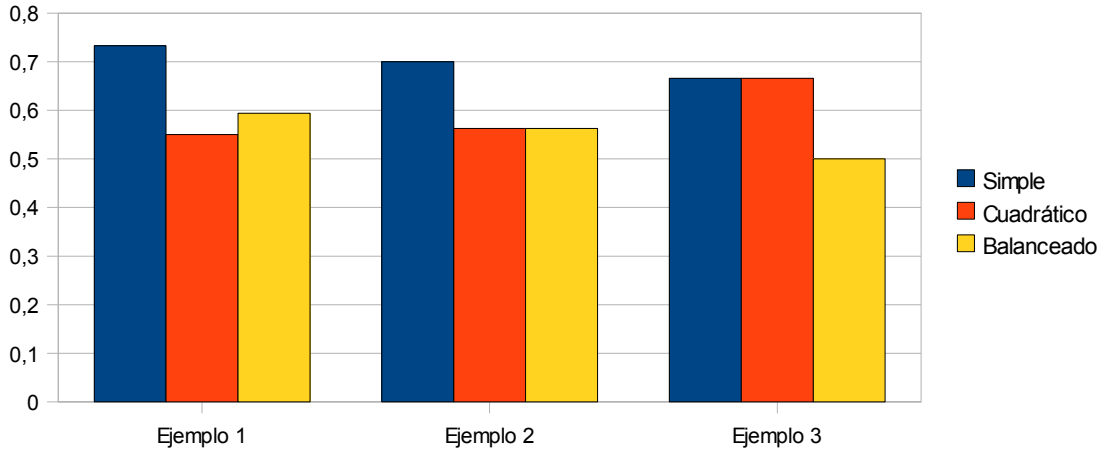
Ejemplo 3

Aceptadas	**Rechazadas**
1	0
1	0
1	
1	

Resultados:

	Simple	**Cuadrático**	**Balanceado**
Ejemplo 1	0,733	0,550	0,594
Ejemplo 2	0,700	0,563	0,563
Ejemplo 3	0,666	0,666	0,500

Gráficamente:



Como se puede apreciar, los calculadores pueden dar valores muy diferentes según las circunstancias del problema y esto será muy importante para los resultados del algoritmo.

El calculador simple tiende a ofrecer resultados más generosos con la bondad de los individuos de la población.

El calculador cuadrático sin embargo, funciona igual sólo si las probabilidades son 0 y 1, pero devuelve valores menores en posiciones intermedias, favoreciendo la búsqueda de AFPs que sean mejores en conjunto.

El calculador balanceado funciona de forma similar al cuadrático, dando los mismo valores que éste cuando hay igual número de palabras rechazadas y aceptadas. Sin embargo, vemos que los resultados no son siempre iguales porque cuando dichos números no son iguales, tiende a dar valores que representan mejor el funcionamiento del autómata.

3.5.4 "Resolutores"

Llamamos "resolutor" a la pieza que resuelve el problema de calcular la probabilidad de aceptación de una palabra en un AFP. Cabe señalar que esto es algo específico de nuestro proyecto y está indirectamente relacionado con el algoritmo genético implementado, sin embargo, como es un parámetro que configuramos junto con los demás, éste parece el lugar más indicado para describirlos. Además, el uso de un "resolutor" sobre el otro no afecta los resultados (se ha probado extensivamente esto) y por lo tanto sólo afecta al rendimiento del algoritmo.

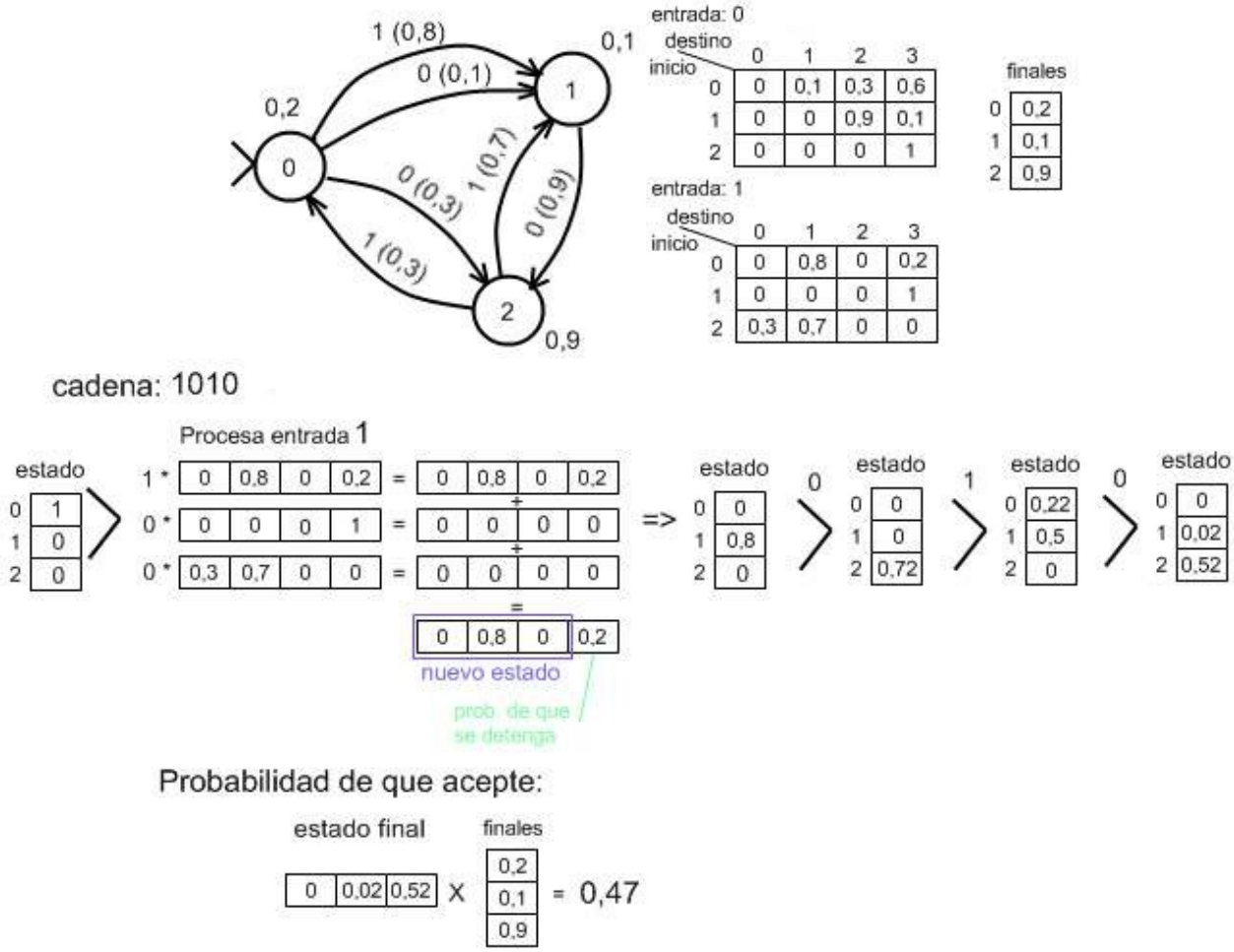
"Resolutor" de AFP por pila

Este "resolutor" realiza su función mediante una pila en la que se van apilando estados. Las probabilidades de cada transición se van multiplicando para calcular la probabilidad final de que una palabra sea aceptada.

Este "resolutor" fue el primero implementado y tiene la ventaja de que es más generalizable. Sin embargo, no es muy utilizado dado que es muy lento, especialmente comparado con el que se explica a continuación.

"Resolutor" de AFP por vectores

Este "resolutor" fue implementado más tarde, aprovechando las características especiales de los AFPs, de forma que es considerablemente más rápido y se basa en guardar los "estados parciales" en vectores. A continuación se muestra gráficamente en un ejemplo como funciona este "resolutor" y se aprovecha la ocasión para mostrar la representación de los AFPs en memoria.



3.6 Tecnología

Lenguaje

Elegimos Java como lenguaje de programación por las ventajas de la programación orientada a objetos y su facilidad de uso. En el servidor, esta elección se debe principalmente a que nos permite utilizar el servidor de aplicaciones JBoss, que es la base de la parte distribuida de la aplicación con el que ya estábamos familiarizados y que permite la utilización de WebServices de forma sencilla. En el cliente, tiene la ventaja de que se puede ejecutar en numerosos sistemas operativos (en teoría todos aquellos que soporten Java, aunque eso no es cierto en la actualidad dado que MacOS X no tiene implementada la versión 6 de la máquina virtual salvo en su última versión). Esta decisión fue respaldada por la realización de un prototipo en C++ como se comenta más adelante.

Control de versiones

Por el tamaño del proyecto y porque realizamos el desarrollo en equipo, nos ha resultado fundamental usar un sistema de control de versiones. Entre los distintos sistemas de control de versiones que existen, nosotros elegimos utilizar CVS, por su facilidad y porque nos ofrecía más que suficientes opciones y ventajas para mantener actualizado y controlado todo el código de nuestro proyecto.

Hemos mantenido nuestro proyecto alojado en Berlios (http://www.berlios.de/). Se trata de una plataforma que ofrece a desarrolladores y programadores repositorios gratuitos para alojar sus proyectos. Entre muchas otras opciones, ofrece CVS, y lo elegimos porque su uso resulta sencillo.

También cabe mencionar que utilizamos Google Docs para escribir la primera versión de la documentación, y éste cuenta con un sistema de control de versiones muy avanzado que nos permite disponer constantemente de la última versión actualizada de los documentos, editar simultáneamente entre todos los miembros del grupo un mismo documento y volver a versiones anteriores de todos los documentos.

Entorno

El entorno de programación que utilizamos es Eclipse. La facilidad de uso de Eclipse para manejar el sistema de control de versiones fue una razón importante para elegirlo como plataforma de trabajo, pero además este sistema es gratuito, dispone de un gran número de plugins para añadir distintas funcionalidades y posee un depurador muy cómodo y avanzado que son fueron de gran utilidad y facilitaron el desarrollo del proyecto.

Plataformas

Una parte importante del sistema se ha desarrollado con la plataforma J2EE, mediante la utilización de EJB3 (Enterprise Java Beans 3.0), necesario para la parte distribuida del sistema. Estas tecnologías son muy utilizadas en la actualidad para el desarrollo de proyectos comerciales.

Parte distribuida

Para implementar la parte del servidor, elegimos JBoss como servidor de aplicaciones. JBoss es un servidor de aplicaciones gratuito con un implementación completa de J2EE. Para compilar el código

generado para el servidor, utilizamos Maven, que también es una aplicación gratuita. Estas dos herramientas están siendo ampliamente utilizadas en entornos profesionales, por lo que creemos que la experiencia adquirida será muy útil para nuestra carrera.

Entre las funciones utilizadas del JBoss, podemos destacar el conector con la base de datos MySQL y el sistema JBossWS para publicar WebServices a partir del código por medio de anotaciones. Al utilizar EJB3 también aprovechamos sus anotaciones y la filosofía POJO (Plain Old Java Object), que intenta mantener un código libre de dependencias y código propio de la arquitectura para simplificar el desarrollo y el mantenimiento.

Los clientes para los WebServices se crearon por medio del plugin para Maven de CXF (de Apache). También realizamos las pruebas de crearlos mediante JaxWS (de JBoss), pero esto no presenta ninguna ventaja que pudiéramos apreciar, por lo que nos decantamos por la primera de estas opciones, aunque incluimos los ficheros pom (de Maven) para generarlos de cualquiera de las dos formas.

El servidor está alojado en una máquina virtual utilizando el software gratuito "Virtual PC" de Microsoft.

Base de datos

La base de datos alojada en el servidor utiliza MySQL, también gratuito. MySQL tiene todas las ventajas de las bases de datos profesionales como Oracle.

Prototipo en C++

Aunque teníamos bastante claro que Java era una buena elección como lenguaje para implementar el proyecto, tuvimos la oportunidad de realizar un prototipo en C++. Este se basó en una parte del código del primer prototipo transformada, capaz de aplicar el algoritmo genético a partir de una serie de ejemplos de cadenas aceptadas y rechazadas para obtener un autómata.

Al realizarlo, pudimos aprovechar las ventajas de este lenguaje como optimizaciones a nivel de ensamblador y control directo de la memoria. Conseguimos que funcionara rápidamente, más rápido que el prototipo en Java, pero sin embargo la complejidad para conseguirlo fue mayor.

La realización de este prototipo fue suficiente para descartar la posibilidad de utilizar C++ para el resto del desarrollo, porque sabíamos *a priori* que al introducir la comunicación por Internet y demás funcionalidad que teníamos planeada la complejidad de hacerlo en Java sería mucho menor. Además, el tiempo de ejecución en C++ no era sustancialmente menor y no compensaba por los demás inconvenientes.

3.6.1 Resumen de tecnologías utilizadas

- Lenguajes: Java, C++
- Control de versiones: CVS
- Entorno: Eclipse
- Plataformas: J2SE, J2EE, EJB3, JBoss
- Base de datos: MySQL
- Gestores de compilación: Maven

- Documentación: Google Docs
- Frameworks: Swing
- Plugins: Maven CFX (Apache), Eclipse Metrics (Eclipse), Maven plugin for Eclipse
- Máquina virtual para creación del servidor: Virtual PC (Microsoft)
- Anotaciones: EJB3 y JBossWS
- Filosofía: POJO

3.7 El sistema desarrollado

3.7.1 Introducción

El sistema ha sido desarrollado en Java, utilizando las tecnologías comentadas en la sección Tecnología. Como ya hemos comentado, dispone de dos aplicaciones: la aplicación cliente y la aplicación administrador. La cantidad de código implementado asciende a más de 8000 líneas de código.

Optamos por un diseño organizado y con un árbol de paquetes bien estructurado.

En la implementación de nuestro sistema aparecen diferenciadas varias partes, que son organizadas en paquetes:

- *Interfaz*. Paquete que contiene todo lo relacionado con la interfaz gráfica de la aplicación.
- *ProblemaAFP*. Paquete que contiene todo lo relacionado con la resolución del problema con autómatas finitos probabilistas.
- *WebServices*. Paquete que contiene las clases necesarias para hacer funcionar los WebServices y permitir la conexión con la base de datos del servidor.
- *GeneradorAutomatico*. Paquete que contiene las clases relacionadas con la creación de problemas automáticos.

3.7.2 Organización de los paquetes

Paquete principal

Las clases principales que se ejecutarán:

- *PrincipalAdministrador*. Esta es la clase que lanza la aplicación del administrador
- *PrincipalCliente*. Esta es la clase que lanza la aplicación del cliente.

Interfaces importantes:

- *Cruzador*. Interfaz de los cruzadores.
- *Individuo*. Interfaz de los individuos utilizados en el algoritmo genético.
- *Mutador*. Interfaz de los mutadores.
- *PoblacionInicial*. Interfaz para la población inicial.
- *Selector*. Interfaz del selector de individuos.

Otras clases importantes:

- *Algoritmo*. Clase que ejecuta el algoritmo genético.
- *Poblacion*. Clase que representa a la población.

Paquete Interfaz

El paquete Interfaz contiene todo lo relacionado con la interfaz gráfica. Contiene a su vez otros

subpaquetes. En el paquete principal de la interfaz están los elementos principales de las interfaces. En los subpaquetes, aparecen frames, componentes y menús que se utilizan en los interfaces principales.

El paquete principal contiene:

- *InterfazAdministrador.* Clase de la interfaz gráfica de la aplicación del administrador.
- *InterfazCliente*. Clase de la interfaz gráfica de la aplicación del cliente.
- *InterfazGrafica*. Interfaz que implementan *InterfazAdministrador* e *InterfazCliente*.
- *ResolverProblemas*. Clase principal de la interfaz gráfica desde la que se resuelven los problemas.

Subpaquete Interfaz.componentes

Este subpaquete del paquete interfaz contiene los elementos necesarios que deben definirse para representar un autómata.

- *AF.* Clase que representa un autómata finito.
- *Estado*. Clase que representa un estado de un autómata.
- *Transición*. Clase que representa una transición de un autómata.

Subpaquete Interfaz.data

Este subpaquete contiene todo lo relacionado con los datos: soluciones, problemas...

- *Configuracion.* Clase que contiene los datos concernientes a las configuraciones de los problemas.
- *Problema*. Clase que representa un problema de la base de datos.
- *Solucion*. Clase que representa una solución de un problema.
- *Stats*. Clase que contienen las estadísticas y datos de los problemas.

Subpaquete Interfaz.frames

Este subpaquete contiene distintos frames utilizados en la aplicación.

- *FrameCadenas.* Clase que muestra el frame que aparece al manipular las cadenas.
- *FrameCargarProblema*. Clase que muestra el frame que aparece al gestionar los problemas de la base de datos.
- *FrameConfiguraciones*. Clase que muestra el frame que aparece al modificar configuraciones.
- *FrameEstadisticasAvanzadas*. Clase que muestra el frame que aparece al mostrar las estadísticas avanzadas.
- *FrameResolver.* Clase que muestra el frame que aparece al resolver el problema actual cargado.
- *FrameResolverCadenas*. Clase que se encarga de resolver un problema a partir de una lista

de cadenas.

- *FrameSoluciones*. Clase que muestra el frame que aparece al Explorar las soluciones de la base de datos.
- *FrameStats*. Clase que muestra las estadísticas básicas.

Subpaquete Interfaz.menus

Este subpaquete contiene los distintos menús que aparecen la interfaz gráfica.

- *MenuAdministrador*. Clase que muestra el menú que aparece en la interfaz del administrador.
- *MenuCliente*. Clase que muestra el menú que aparece en la interfaz del cliente.

Subpaquete Interfaz.paneles

Este subpaquete contiene 17 paneles que son utilizados en la aplicación.

Paquete GeneradorAutomatico

- *GeneradorAF*. Clase que se encarga de generar autómatas automáticamente.
- *GeneradorCadenas*. Clase que se encarga de generar cadenas automáticamente.

Paquete ProblemaAFP

- *AFP*. Clase que define el objeto de un autómata finito probabilista.
- *AFPIniciales*. Clase que implementa *PoblacionInicial*. Contiene los individuos de la población inicial.
- *AplicarAlgoritmoAFP*. Clase que prepara los parámetros para llamar al algoritmo genético.
- *CalculadorBondad*. Clase que calcula la bondad de un AFP.
- *Estado*. Clase utilizada para guardar los datos de los estados de un AFP al aplicar el algoritmo genético.
- *GeneradorAleatorioAFP*. Clase que genera probabilidades aleatorias para un AFP.
- *ParametrosAFP*. Clase utilizada para definir los parámetros de un AFP.
- *ProblemaAFP*. Clase que representa un problema.
- *ResolverAFP*. Interfaz que será implementada por otras clases, para resolver un problema.
- *SelectorAFP*. Clase que implementa *Selector*, y se encarga de seleccionar los mejores AFPs en un problema.
- *Transición*. Clase que representa las transiciones de un AFP.

El paquete *ProblemaAFP* contiene también otros subpaquetes, donde aparecen las clases de implementaciones concretas de elementos del algoritmo genético, y las factorías de estos elementos:

Subpaquete Factorías

Contiene varias factorías que construyen un tipo concreto de parámetro de configuración.

- *CalculadorBondadFactory.*
- *CruzadorAFPFactory.*
- *MutadorAFPFactory.*
- *ResolverAFPFactory.*

Subpaquete CalculadoresBondad

Contiene 4 tipos distintos de calculadores de bondad.

- *CalculadorBondadSimple*.
- *CalculadorBondadCuadratico*.
- *CalculadorBondadBalanceado*.
- *CalculadorBondadPreferenciaDet*.

Subpaquete Cruzadores

Contiene 4 cruzadores diferentes, además de un cruzador nulo, que no cruza los individuos.

- *CruzadorAFP_1*
- *CruzadorAFP_2*
- *CruzadorAFP_3*
- *CruzadorAFP_4*
- *CruzadorAFPNulo*

Subpaquete Mutadores

Contiene 3 mutadores diferentes, además de un mutador nulo, que no muta los individuos.

- *MutadorAFP_1*
- *MutadorAFP_2*
- *MutadorAFP_3*
- *MutadorAFP_4*
- *MutadorAFPNulo*

Subpaquete Resolver

Contiene dos clases que implementan dos formas distintas de resolver el problema:

- *ResolverAFPPila*. Clase que resuelve la aceptación de una palabra en un AFP mediante una pila.
- *ResolverAFPVectores*. Clase que resuelve la aceptación de una palabra en un AFP mediante

vectores.

Paquete util

Este paquete contiene tan sólo una clase necesaria para conectar con el servidor, y que no encaja en otro paquete. Se trata de la clase Config.

Paquete WS

Este paquete contiene todas las clases que realizan operaciones relacionadas con la parte de los *WebServices*.

- *AddProblemaWS*. Clase que añade un nuevo problema a la base de datos.
- *GetAdvancedStatsWS*. Clase que calcula las estadísticas avanzadas.
- *GetProblemBasicWS.* Clase que recoge un problema para llamar al método que ejecutará el algoritmo.
- *GetProblemasWS.* Clase encargada de capturar una lista de todos los problemas de la base de datos.
- *GetProblemaWS*. Clase que se encarga de coger los datos de un problema concreto de la base de datos.
- *GetSolucionesWS*. Clase que se encarga de recoger todas las soluciones de la base de datos.
- *GetSolucionWS*. Clase que se encarga de extraer los datos concretos de una única solución de la base de datos.
- *GetValidValuesWS*. Clase que obtiene los valores válidos para una configuración.
- *RemoveProblemaWS*. Clase que se encarga de eliminar un problema de la base de datos.
- *SetSolucionWS*. Clase que se encarga de guardar una solucion en la base de datos.

3.7.3 Patrones de diseño empleados

El sistema dispone de varios patrones de diseño aplicados en distintas partes del código.

Patrón Factory:

El patrón Factory es utilizado en varias partes del código. Se dispone de factorías de todos los elementos configurables de un problema de AFPs: cruzadores, mutadores, calculadores de bondad y resolutores.

Patrón Iterator:

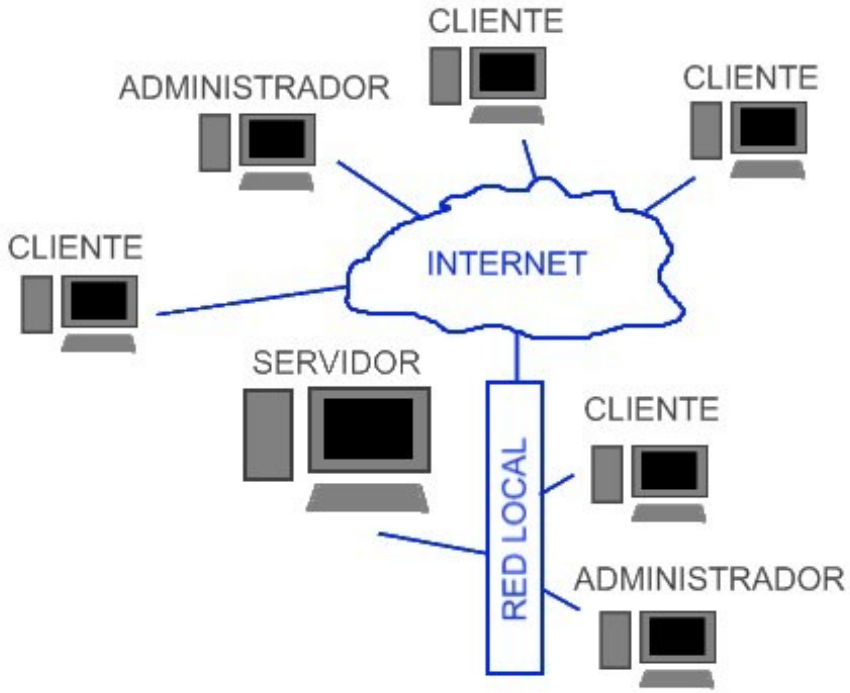
En distintas partes del código, donde hay que recorrer estructuras se utiliza el patrón Iterator.

Patrón Singleton:

Este patrón permite obtener estáticamente una instancia de una clase. Es utilizado por ejemplo para pasar los parámetros de configuración a las distintas partes del algoritmo genético.

3.7.4 Servidor

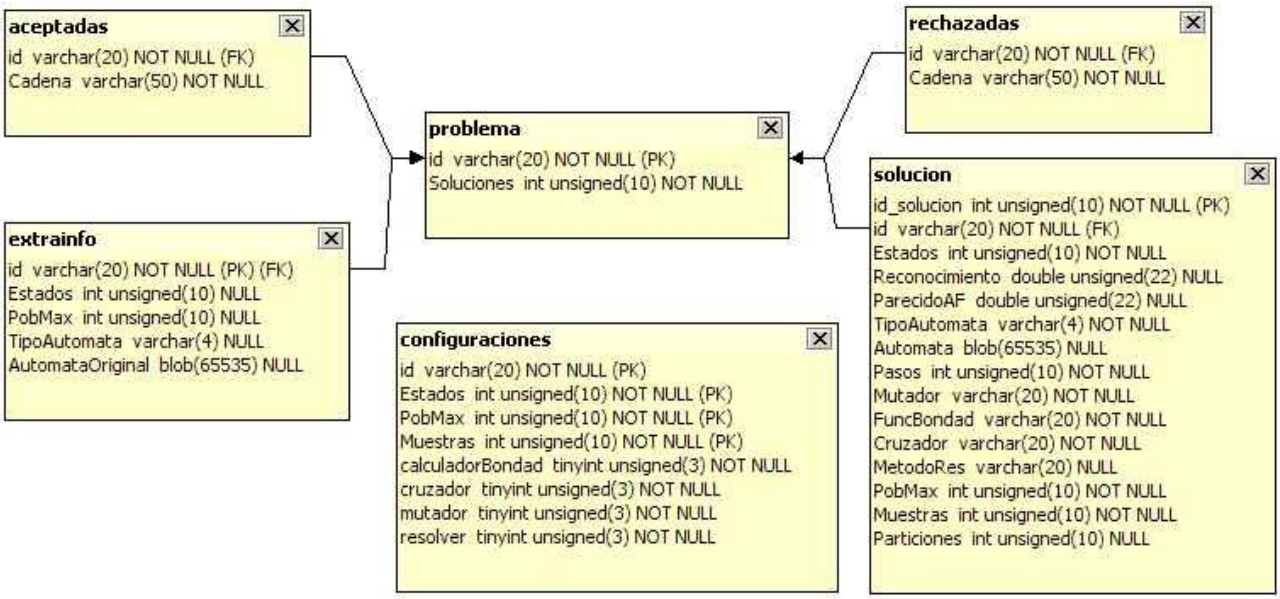
El servidor permite que múltiples clientes resuelvan los mismos problemas y los resultados se almacenan en una base de datos común para posteriormente ser analizados. Además, permite añadir problemas nuevos y consultar distintas estadísticas de forma remota, facilitando el uso y administración del sistema.



Entorno

Utilizamos el servidor de aplicaciones JBoss para desplegar el servidor. JBoss es un servidor de aplicaciones gratuito y de código abierto (pertenece a RedHat) que tiene muchas características que lo hacen muy útil para nuestra aplicación. En particular, hemos utilizado JBossWS, que es una extensión del JBoss incluida en la distribución (aunque tuvimos que actualizarla, pero eso se detalla más adelante) que permite publicar WebServices de manera muy sencilla y sólo creando código Java.

Los problemas y soluciones se incluyen en una base de datos MySQL (nuevamente una tecnología gratuita, aunque no de código abierto). La base de datos utilizada tiene la siguiente forma:



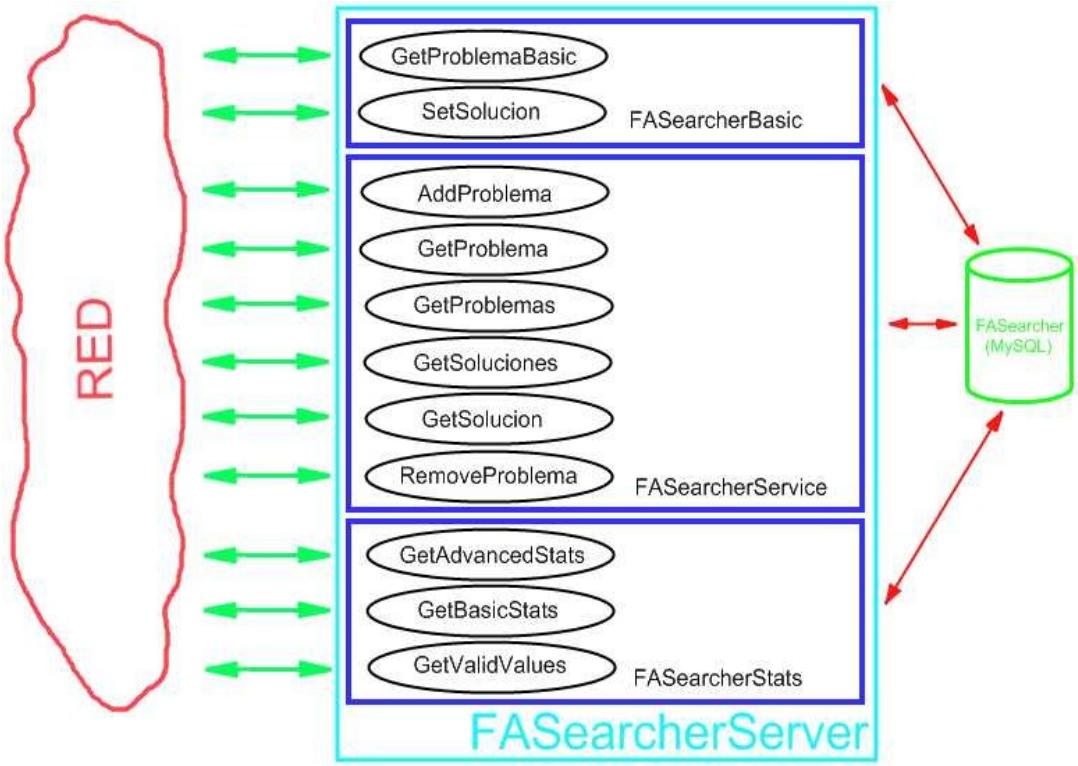
En el entorno de desarrollo, aunque no es necesario para ejecutar el servidor, se incluyó también el sistema Maven (también gratuito, perteneciente a Apache) para compilar el código. Este es un sistema para compilar código muy completo, se considera una evolución del sistema "Ant" (muy utilizado, por ejemplo en Linux). Maven permite no sólo compilar el código sino también realizar distintos tipos de empaquetados (jar, sar, war, ear, etc.), llamar a otros sistemas complejos antes y después de la compilación en sí (crear descriptores de servicios web, realizar pruebas sobre el código, etc.) e importar directamente las librerías necesarias desde repositorios locales o de Internet. Además, los ficheros que describen lo que hacer con el código en Maven tienen formato XML lo que facilita su lectura, les da mayor modularidad y permite comprobaciones sintácticas mientras se escriben, dándole otra ventaja frente a los ficheros de textos sin formato de "Ant".

El servidor final, accesible a los usuarios de la aplicación, está creado como una máquina virtual utilizando Virtual PC de Microsoft (que ofrece gratuitamente a los usuarios de Windows) con una versión de Windows XP instalada. De esta forma se aisla el servidor de otros procesos que puedan estar funcionando en el ordenador anfitrión y proporciona mayor seguridad. Además, se pueden realizar copias de seguridad del sistema entero y se puede cambiar el servidor a otro anfitrión si fuera necesario de manera fácil y rápida, sin apenas realizar configuraciones. Además se creó un DNS dinámico (en DynDNS.org, siendo este un servicio ofrecido de forma gratuita) para poder cambiar la dirección IP del servidor sin problemas.

Código

El código desarrollado está en forma de JavaBeans con anotaciones de EJB3 y de JBossWS, utilizadas por el servidor de aplicaciones para desplegar el sistema correctamente. Esta forma de programar crea código muy simple de Java (frecuentemente denominado POJOs, Plain Old Java Objects) facilitando su comprensión y mantenimiento.

Se crearon dos EJBs, para los dos servicios que ofrece el servidor. Uno de ellos es el usado por los "clientes", que sólo puede solicitar problemas y devolver su solución. El otro servicio está enfocado a administradores, facilitando el crear nuevos problemas en la base de datos.



La conexión con la base de datos se establece directamente mediante JBoss, que da acceso a un DataSource.

Compilación

Para poder compilar el código, es necesario disponer del sistema Maven (http://maven.apache.org/download.html) instalado en el ordenador. Se deben seguir todos los pasos indicados, incluidos el de añadir el directorio bin de la instalación al PATH de Windows (también se podría compilar en Linux y el procedimiento es el mismo). También es necesario, al menos la primera vez, tener acceso a Internet para poder resolver las dependencias. Maven guarda luego las librerías necesarias en el repositorio local, por lo cual sólo las descarga de Internet una vez.
Una vez instalado Maven, hay que desplegar el proyecto en el repositorio local, para que se puedan resolver las dependencias que tiene el servidor con éste en tiempo de compilación. Para esto tenemos que ejecutar el siguiente comando reemplazando [fichero ProgramadorGenetico.jar] por la dirección del fichero:

mvn install:install-file -DgroupId=es.si -DartifactId=ProgramadorGenetico -Dversion=0.2 -Dpackaging=jar -Dfile=[fichero ProgramadorGenetico.jar]

Ahora ya estamos listos para compilar. Para esto se debe ir al directorio raíz del proyecto del servidor (donde se encuentra el fichero pom.xml, que es el descriptor de maven) y ejecutar el siguiente comando:

mvn clean install

Esto nos creará el fichero jar en el directorio target correspondiente (“%raiz

%\target\FASearcherServer.jar"). Este es el fichero que luego desplegaremos en JBoss.

Desplegado

Para desplegar el servidor en un ordenador es necesario disponer del JBoss (http://labs.jboss.com/jbossas/downloads/). En particular utilizamos la versión 4.2.2.GA que es la última versión (no beta) disponible en el momento de crearlo. Además, se debe añadir a este la ultima versión de los JBossWS (http://labs.jboss.com/jbossws/downloads/) que no viene incluida por defecto pero es indispensable para que funcione con Java 1.6 (la versión que estamos utilizando a lo largo del proyecto).

El JBoss no necesita instalación, sólo descomprimir la carpeta. Para iniciar el servicio se utiliza el fichero run.bat incluido en el directorio bin de la instalación. En particular, el comando a ejecutar es:

```
run.bat -b 0.0.0.0
```

para ligarlo a todas las interfaces disponibles en el ordenador. Por defecto sólo se conecta una conexión loopback accesible en la máquina donde está instalado. Además, en nuestro caso modificamos el fichero server\default\deploy\jboss-web.deployer\server.xml para que utilice el puerto 18080 en lugar del 8080 establecido por defecto.

Como se mencionó antes, utilizamos una base de datos MySQL, a la que también hay que tener acceso desde el entorno de desplegado. En proyecto del servidor, en la carpeta resources, se incluye un script sql para la creación de todas las tablas necesarias. También es necesario configurar JBoss para que utilice esta base de datos, modificando los parámetros del fichero: server\default\deploy\mysql-ds.xml. Cabe destacar que este fichero no se encuentra ahí en la configuración por defecto del JBoss, por lo cual antes hay que seguir los pasos de la documentación que indican cómo utilizar bases de datos MySQL.

Otro problema que tuvimos es que el puerto en los WSDL (descriptores de servicios) quedaba ligado por defecto al identificador del ordenador en la red local, haciendo imposible el acceso desde fuera de esta red.

Para esto hay que modificar el fichero server\default\deploy\jbossws.sar\jbossws.beans\META-INF\ jboss-bean.xml, indicando para la propiedad "webServiceHost" el valor "fasearcher.selfip.info" (sin las comillas) y no la dirección de ligadura por defecto del jboss.

Por último, en la máquina anfitrión se debe crear un túnel para el puerto 18080, esto se consigue mediante el software fpipe (distribuido de forma gratuita por McAfee), ejecutando la siguiente instrucción:

```
fpipe.exe -l 18080 -s 18080 -r 18080 [ip_maquina_virtual]
```

3.7.5 Interfaz gráfica

Hemos decidido realizar dos interfaces gráficas independientes. Una para los clientes, donde pueden ver lo que esta haciendo el sistema mientras resuelven problemas y otra para la administración del servidor, con el fin de añadir problemas y consultar estadísticas.

Interfaz cliente

Esta interfaz es la que ve cualquier persona que desea colaborar con nuestro proyecto (y nosotros mismos cuando dedicamos nuestros recursos a resolver problemas).

Las funcionalidades de esta interfaz son:

- Permitir que el usuario colabore con su ordenador resolviendo problemas planteados por el servidor
- Hacer el entorno más agradable al usuario
- Permitir ver en mayor detalle lo que está haciendo el sistema, aumentando la confianza y el interés para el usuario
- Permitir ver los pasos que llevan de autómatas aleatorios a autómatas que se ajustan a los patrones del problema

Menús de la interfaz:

Acciones	**Vista**
Resolver un problema	Mostrar autómatas generados (booleano)
Empezar a resolver problemas Parar de resolver problemas	

Cuando se resuelve un solo autómata, se pueden ver los pasos realizados. Cuando se resuelven muchos, sólo se ve el final a medida que se genera. Esto sucede cuando está seleccionada la opción de "Mostrar autómatas generados". En caso contrario, el usuario no ve nada del proceso salvo el número de autómatas resueltos.

Interfaz servidor

Esta interfaz es la que utilizaremos principalmente nosotros, con el fin de poder añadir nuevos problemas a la base de datos para realizar pruebas. Además, esta interfaz tendrá funcionalidades de análisis avanzadas, para ver estadísticas de problemas particulares o consultar soluciones determinadas.

Las funcionalidades de esta interfaz son:

- Facilitar la creación de nuevos problemas (tanto de forma manual como aleatoria, siempre con un importante componente gráfico)
- Abstraer la interacción con el servidor, facilitando la posible incorporación de nuevos administradores no familiarizados con los detalles de implementación

- Proporcionar consultas preestablecidas con la base de datos para minimizar errores y agilizar el trabajo (ver estadísticas filtrando ciertas características de los problemas)
- Poder mostrar de forma sencilla y gráfica todas las funcionalidades incorporadas en nuestro sistema a terceros

Menús de la interfaz:

<table>
<tr><th>Problema</th><th>Soluciones</th><th>Acciones</th><th>Estadísticas</th><th>Ayuda</th></tr>
<tr><td>Gestionar Problemas</td><td>Explorar soluciones</td><td>Resolver problema actual</td><td>Ver estadísticas básicas</td><td>Sobre la aplicación</td></tr>
<tr><td>Dibujar autómata

Generar autómata aleatorio</td><td rowspan="3"></td><td>Resolver problema desde cadenas</td><td>Ver estadísticas avanzadas</td><td rowspan="3"></td></tr>
<tr><td>Generar cadenas aleatorias
Manipular cadenas
Modificar configuraciones</td><td colspan="2"></td></tr>
<tr><td>Enviar problema</td><td colspan="2"></td></tr>
</table>

La opción de enviar problema sólo se activa cuando el usuario ya creó un autómata y estableció las cadenas aceptadas y rechazadas. Estamos obligando a crear el autómata en la interfaz, no permitimos poner directamente las cadenas aceptadas y rechazadas, que para nosotros tiene sentido porque nos puede permitir analizar en más detalle los resultados (comparando autómata original con autómata creado).

4 Pruebas planteadas

4.1 Introducción

Una parte clave del desarrollo de este proyecto concierne a las pruebas que hemos podido realizar sobre este sistema.

Los objetivos principales de estas pruebas son:

- Comprender cuáles son las principales virtudes y defectos de resolver este problema utilizando un algoritmo genético que utiliza AFPs.
- Extraer conclusiones sobre cuáles son los mejores parámetros para resolver problemas y sobre la cantidad de cadenas necesarias o adecuadas para resolver problemas.
- Obtener una gran cantidad de soluciones a distintos problemas utilizando configuraciones diferentes, en las que varían los distintos parámetros del algoritmo definidos anteriormente.

Hemos realizado dos baterías de pruebas durante el proceso de planteamiento y análisis de pruebas.

La primera batería de pruebas consiste en una base de datos que ofrece gran variedad de problemas (30 problemas). Para cada uno de ellos se ofrece una cantidad de configuraciones cercana a 100 para cada problema, muy variada, que ofrece situaciones muy diferentes, con distintos valores de los parámetros del algoritmo.

La segunda batería de pruebas consiste en una base de datos más pequeña, que ofrece menos problemas (11 problemas). Sin embargo, en este caso el número de configuraciones es de 270 configuraciones diferentes para cada problema. Pero estas configuraciones no serán aleatorias, serán fijas para todos los problemas. Esto permitirá un estudio más exhaustivo de la efectividad de configuraciones concretas, dado que conseguimos unos resultados más homogéneos entre problemas.

4.2 Primera batería de pruebas

4.2.1 Objetivos de la primera batería de pruebas

El objetivo de esta batería de pruebas es obtener un gran número de soluciones diferentes para muchos tipos de problemas. A partir de estas soluciones, se extraerán conclusiones de las configuraciones empleadas y las mejores combinaciones de los parámetros para ejecutar el algoritmo. Además, se extraerán datos de calidad globales y datos locales de las mejores configuraciones.

4.2.2 Características de cada problema

Entre las características de cada problema encontramos:

- Tendrá asociados un gran número de configuraciones diferentes. En cada una, variará al menos algún parámetro del algoritmo. Se utilizarán todos los cruzadores, mutadores, y calculadores de bondad definidos. Por tanto, se estarán utilizando 5 cruzadores, 4 mutadores, y 4 calculadores de bondad. Los valores de población máxima utilizados variarán desde 100

hasta 5000, pasando por muchos valores intermedios como: 400, 500, 700, 1000, 2000, 3000 y 4000. Los valores de las muestras empleadas serán: 20,30,40,60.

- El número de configuraciones distintas para cada problema estará aproximadamente entre 70 y 150 (excepto en dos problemas que el número es muy bajo), según el interés que muestra ese problema concreto.
- En ocasiones, se incluirán en la base de datos dos problemas que responden inicialmente al mismo lenguaje deseado (los representa el mismo autómata finito). Sin embargo, serán dos problemas distintos puesto que las cadenas de entrada que representarán al problema serán distintas, o habrá distinto número de ellas en los problemas.

4.2.3 Notación

Las pruebas realizadas respetarán una notación. Cada prueba dispondrá de un identificador que será representado como se describe a continuación.

Separaremos las pruebas en 3 tipos:

- Pruebas básicas
- Pruebas intermedias
- Pruebas avanzadas

Denotaremos las pruebas mediante una letra que identificará el tipo de prueba que es. Las pruebas básicas se identificarán con la letra A, las pruebas intermedias con la letra B y las pruebas avanzadas con la letra C. A esa letra le seguirá un número que será el número de prueba establecido. Así, para las pruebas básicas, habrá prueba A1, prueba A2, prueba A3...

4.2.4 Detalle de las pruebas

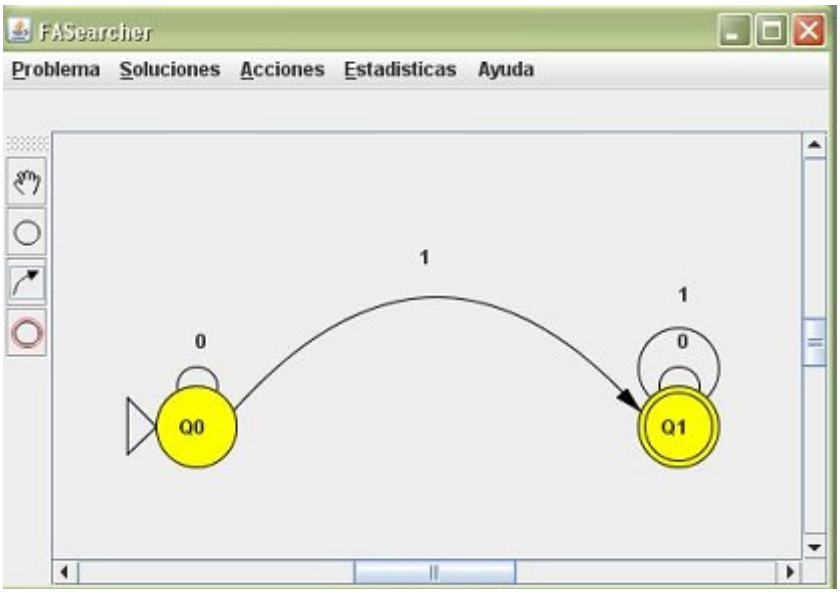
Pruebas básicas (A)

Consideramos conveniente realizar pruebas sobre varios autómatas de pocos estados: 2, 3 y 4 estados, para comprobar la efectividad de la aplicación. Denotamos estas pruebas como pruebas de tipo A.

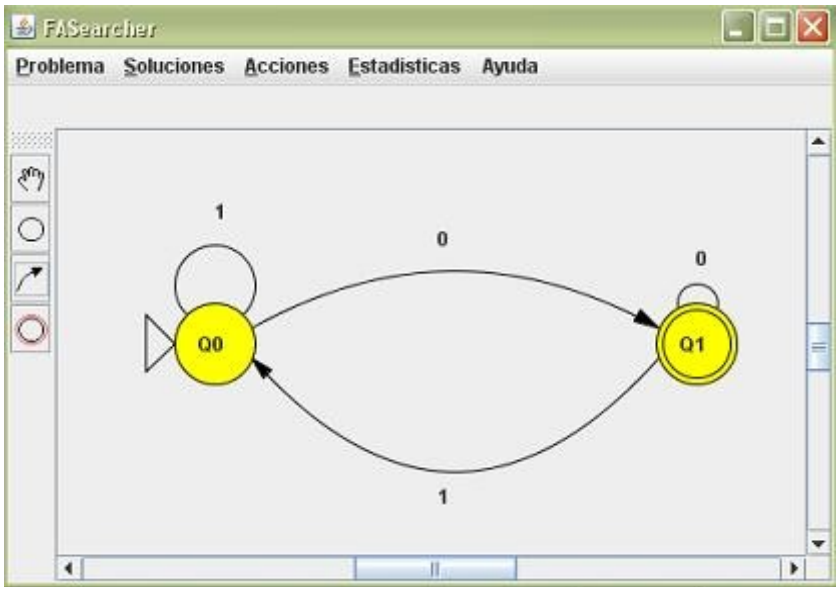
Las pruebas que consideramos:

Nombre de la prueba	Prueba A1
Lenguaje que define el autómata	Lenguaje que acepta las palabras que contienen algún 1.
Número de estados del autómata	2 estados
Número de problemas con el lenguaje	2 problemas
Descripción primer problema	
Número de configuraciones	156
Número de cadenas	20
Descripción segundo problema	

Número de configuraciones	156
Número de cadenas	30



Nombre de la prueba	Prueba A2
Lenguaje que define el autómata	Lenguaje de las palabras que terminan en 0.
Número de estados del autómata	2 estados
Número de problemas con el lenguaje	1 problema
Descripción primer problema	
Número de configuraciones	2
Número de cadenas	12



Nombre de la prueba	Prueba A3
Lenguaje que define el autómata	Lenguaje que acepta las palabras que contienen la secuencia 01.
Número de estados del autómata	3 estados
Número de problemas con el lenguaje	1 problema
Descripción primer problema	
Número de configuraciones	8
Número de cadenas	15

Nombre de la prueba	Prueba A4
Lenguaje que define el autómata	Lenguaje que acepta las palabras que contienen un múltiplo de 3 ceros.
Número de estados del autómata	3 estados
Número de problemas con el lenguaje	2 problemas
Descripción primer problema	
Número de configuraciones	58
Número de cadenas	20
Descripción segundo problema	
Número de configuraciones	58
Número de cadenas	32

Nombre de la prueba	Prueba A5
Lenguaje que define el autómata	Lenguaje que acepta las palabras que contienen tres o más 1's.
Número de estados del autómata	4 estados
Número de problemas con el lenguaje	2 problemas
Descripción primer problema	
Número de configuraciones	96
Número de cadenas	20
Descripción segundo problema	
Número de configuraciones	96
Número de cadenas	32

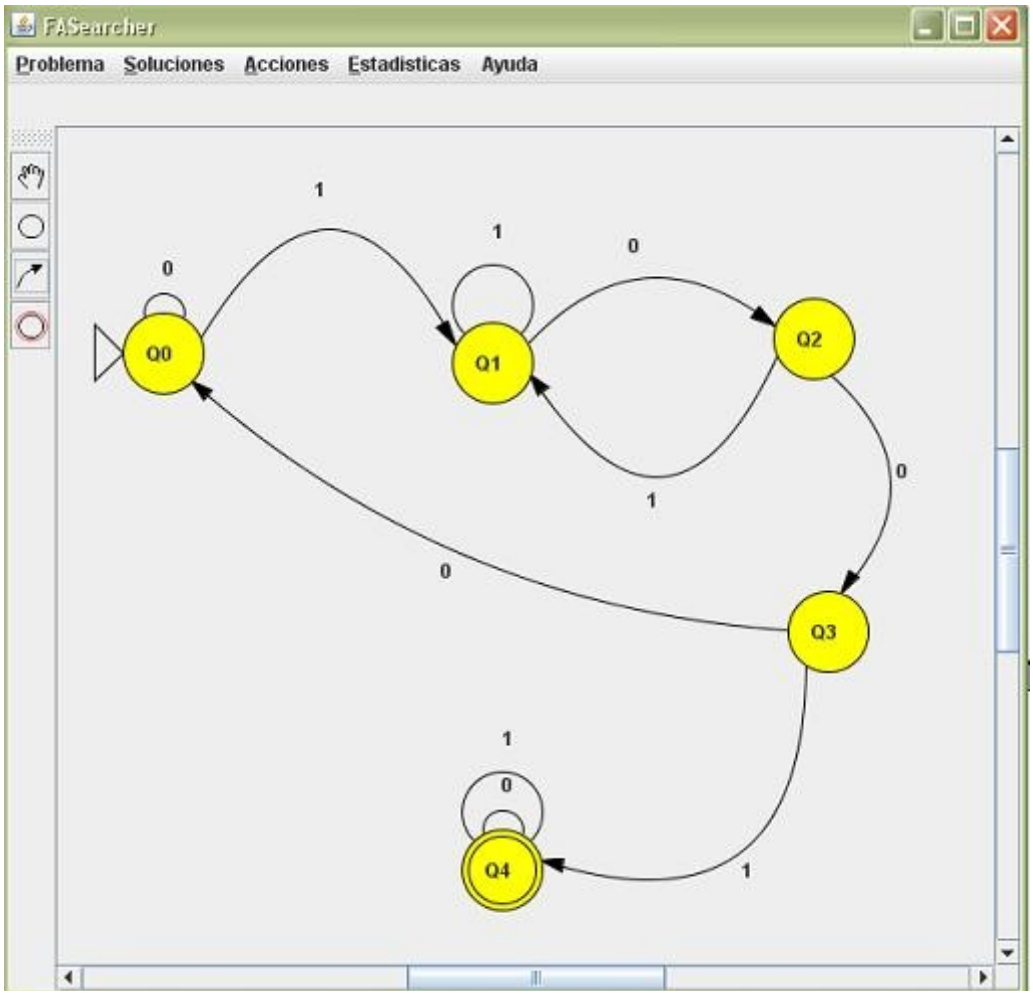
Es importante que los resultados de estas pruebas sean muy positivos, que se obtengan AFPs con una alta precisión, sin transiciones con probabilidades poco definidas. Necesitamos transiciones completamente definidas y no tener errores en lenguajes tan sencillos.

Estas pruebas se realizarán con distintas configuraciones. En estas configuraciones se analizarán todos los cruzadores, mutadores y todos los calculadores de bondad disponibles. Las poblaciones recogerán valores muy dispares, desde 100 hasta 5000 individuos.

Pruebas intermedias (B)

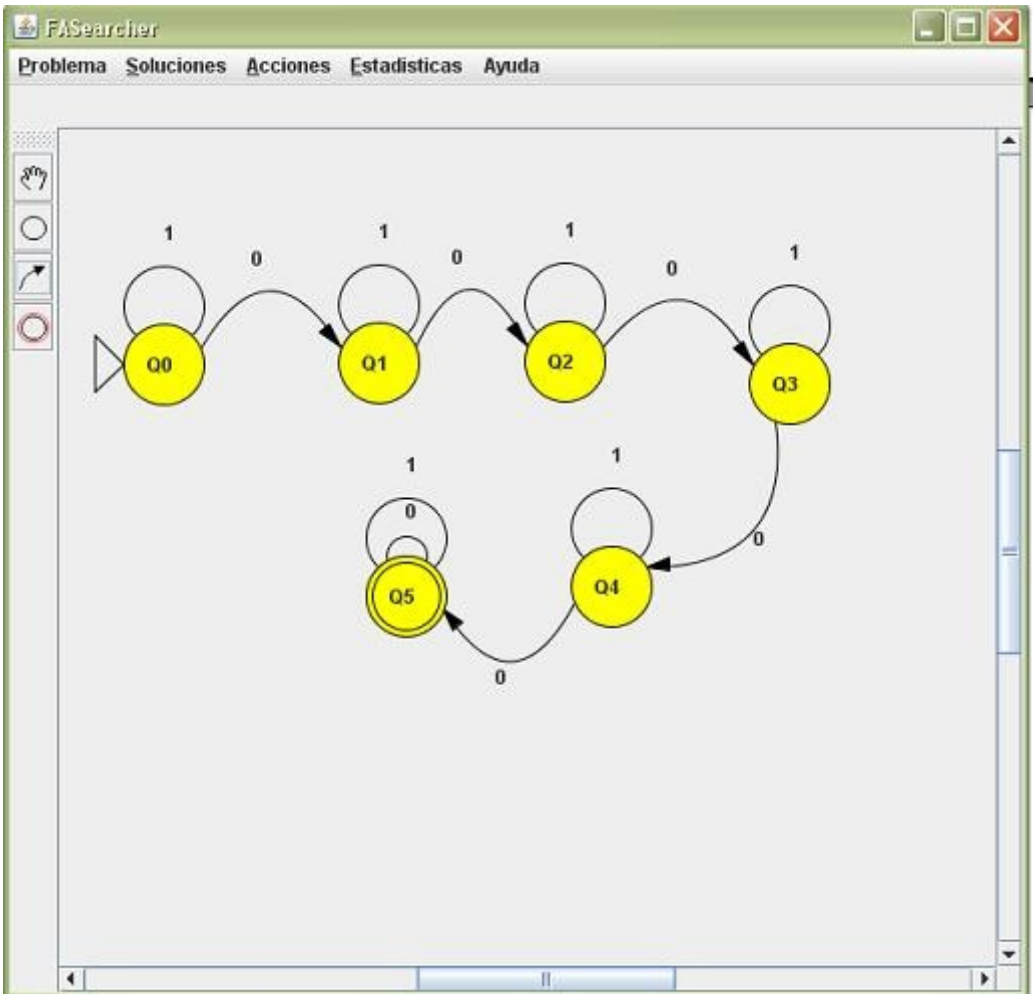
Estas pruebas pueden recoger a aquellos autómatas de un tamaño medio, podemos considerarlos de entre 5 y 8 estados. Podemos crear algunos autómatas manualmente, como por ejemplo:

Nombre de la prueba	Prueba B1
Lenguaje que define el autómata	Lenguaje que acepta las palabras que contengan la secuencia 1001.
Número de estados del autómata	5 estados
Número de problemas con el lenguaje	2 problemas
Descripción primer problema	
Número de configuraciones	88
Número de cadenas	20
Descripción segundo problema	
Número de configuraciones	88
Número de cadenas	30



Nombre de la prueba	Prueba B2
Lenguaje que define el autómata	Lenguaje que acepta las palabras que contengan 5 o más ceros.
Número de estados del autómata	6 estados
Número de problemas con el lenguaje	4 problemas
Descripción primer problema	
Número de configuraciones	118
Número de cadenas	70
Descripción segundo problema	
Número de configuraciones	118
Número de cadenas	50
Descripción tercer problema	
Número de configuraciones	54
Número de cadenas	24

Descripción cuarto problema	
Número de configuraciones	54
Número de cadenas	32



Nombre de la prueba	Prueba B3
Lenguaje que define el autómata	Lenguaje que acepta las palabras que contengan las secuencias 100 o 011.
Número de estados del autómata	6 estados
Número de problemas con el lenguaje	2 problemas
Descripción primer problema	
Número de configuraciones	114
Número de cadenas	40
Descripción segundo problema	

Número de configuraciones	122
Número de cadenas	20

Nombre de la prueba	Prueba B4
Lenguaje que define el autómata	Lenguaje que acepta las palabras que contengan la secuencia 1010101
Número de estados del autómata	8 estados
Número de problemas con el lenguaje	2 problemas
Descripción primer problema	
Número de configuraciones	114
Número de cadenas	60
Descripción segundo problema	
Número de configuraciones	114
Número de cadenas	30

Nombre de la prueba	Prueba B5
Lenguaje que define el autómata	Autómata aleatorio
Número de estados del autómata	5 estados
Número de problemas con el lenguaje	2 problemas
Descripción primer problema	
Número de configuraciones	88
Número de cadenas	40
Descripción segundo problema	
Número de configuraciones	88
Número de cadenas	20

Nombre de la prueba	Prueba B6
Lenguaje que define el autómata	Autómata aleatorio
Número de estados del autómata	6 estados
Número de problemas con el lenguaje	1 problema
Descripción primer problema	
Número de configuraciones	114
Número de cadenas	60

Nombre de la prueba	Prueba B7
Lenguaje que define el autómata	Autómata aleatorio
Número de estados del autómata	7 estados
Número de problemas con el lenguaje	2 problemas
Descripción primer problema	
Número de configuraciones	114
Número de cadenas	70
Configuraciones	Las configuraciones piden 6 estados aunque el autómata original tuviera 7 estados.
Descripción segundo problema	
Número de configuraciones	114
Número de cadenas	50
Configuraciones	Las configuraciones piden 6 estados aunque el autómata original tuviera 7 estados.

Nombre de la prueba	Prueba B8
Lenguaje que define el autómata	Autómata aleatorio
Número de estados del autómata	8 estados
Número de problemas con el lenguaje	1 problema
Descripción primer problema	
Número de configuraciones	114
Número de cadenas	42
Configuraciones	Las configuraciones piden 6 estados aunque el autómata original tuviera 8 estados.

Las configuraciones probadas incluirán muchas posibilidades: cada cruzador, mutador y calculador de bondad se probará en muchas de las configuraciones. Los valores de población y muestras variarán también entre muchos valores.

Como se puede observar , en las pruebas B7 y B8 se intenta ejecutar configuraciones donde el número de estados requerido en las configuraciones es inferior al número de estados del autómata creado aleatoriamente. Con estos problemas se pretende demostrar que el algoritmo encontrará un autómata mínimo si la configuración del algoritmo es adecuada.

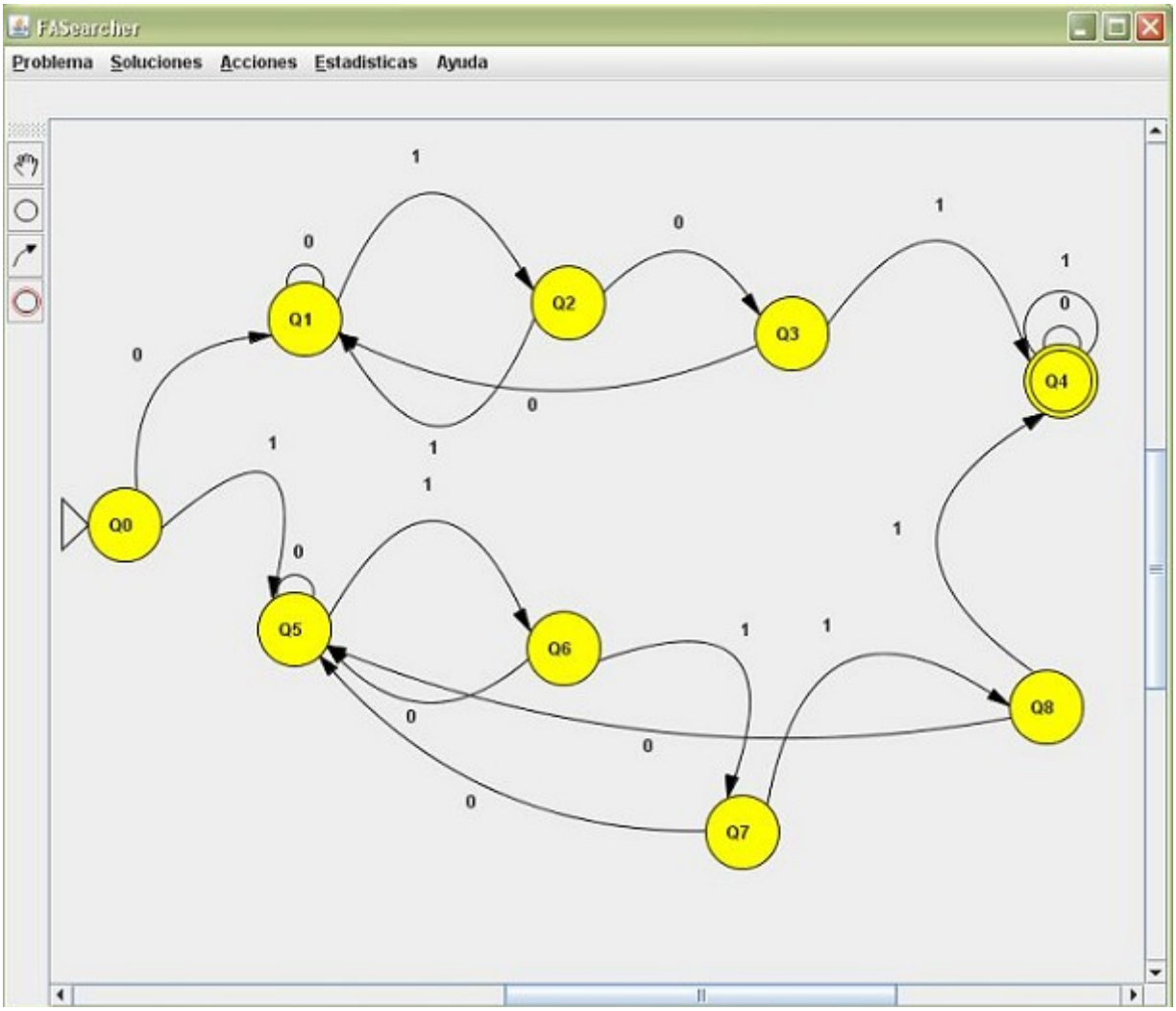
Pruebas avanzadas (C)

Estas pruebas recogen a los demás autómatas a partir de 8 estados. Se espera que el resultado de

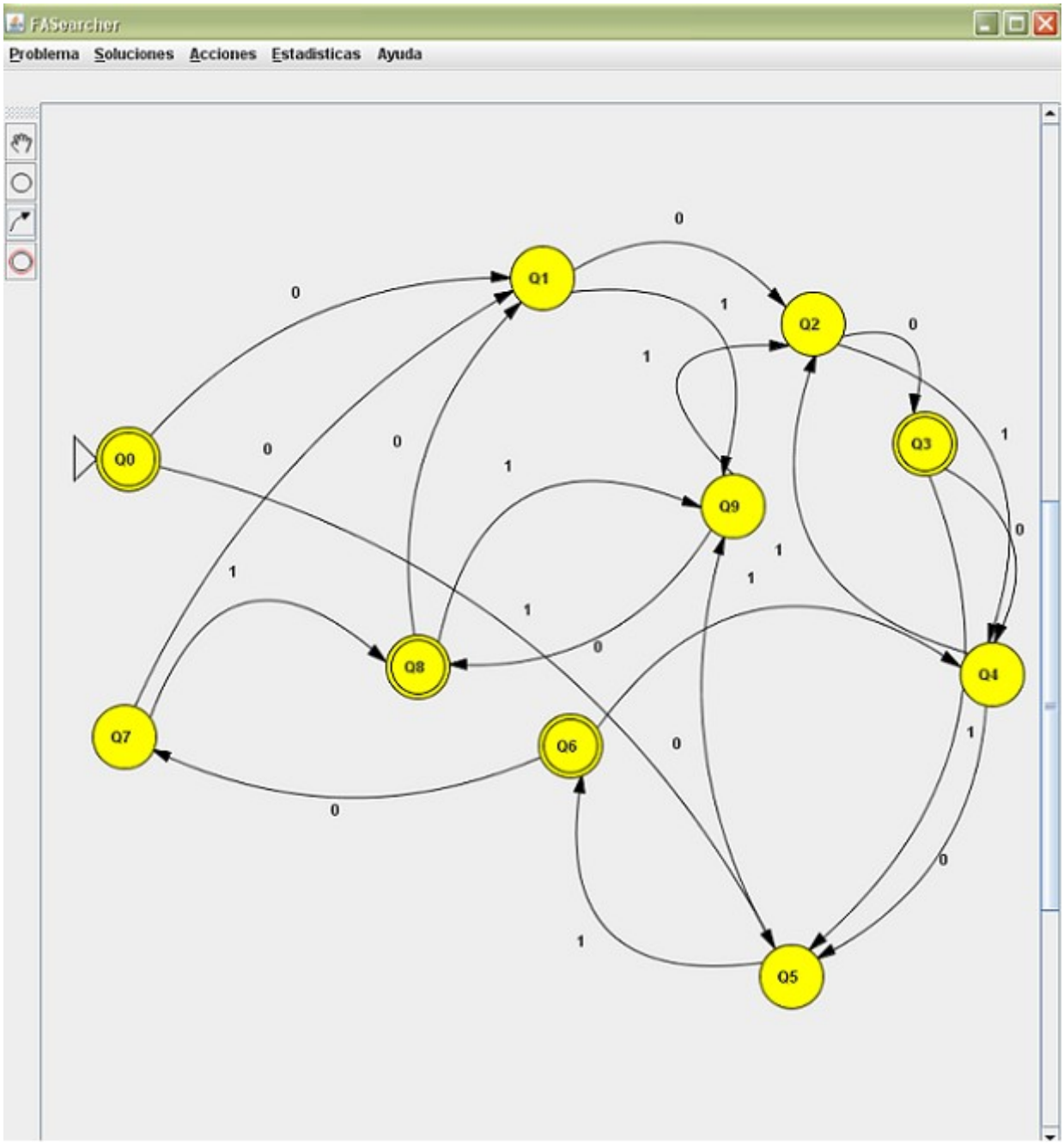
estas pruebas sea peor, puesto que la complejidad del problema aumenta exponencialmente. Además, debemos mencionar que el tiempo de ejecución de estas pruebas también será mayor.

Las pruebas que planteamos inicialmente son:

Nombre de la prueba	Prueba C9
Lenguaje que define el autómata	Lenguaje que acepta: cadenas que empiecen por 0 y contengan la secuencia 101, o cadenas que empiecen por 1 y tengan 4 unos consecutivos.
Número de estados del autómata	9 estados (Autómata mínimo para ese lenguaje)
Número de problemas con el lenguaje	1 problema
Descripción primer problema	
Detalles	En las configuraciones se pide que el autómata tengo 8 o 7 estados como máximo.
Número de configuraciones	25
Número de cadenas	33



Nombre de la prueba	Prueba C10
Lenguaje que define el autómata	Autómata aleatorio
Número de estados del autómata	10 estados
Número de problemas con el lenguaje	2 problema
Descripción primer problema	
Número de configuraciones	183
Número de cadenas	60
Descripción segundo problema	
Número de configuraciones	183
Número de cadenas	40



Nombre de la prueba	Prueba C12
Lenguaje que define el autómata	Cadenas que contienen al menos 6 ceros o 6 unos consecutivos
Número de estados del autómata	12 estados (Autómata mínimo para ese lenguaje)
Número de problemas con el lenguaje	1 problema

Descripción primer problema	
Detalles	Máximo de 10 estados
Número de configuraciones	22
Número de cadenas	70

Nombre de la prueba	Prueba C15
Lenguaje que define el autómata	Autómata aleatorio
Número de estados del autómata	15 estados
Número de problemas con el lenguaje	2 problema
Descripción primer problema	
Número de configuraciones	100
Número de cadenas	40
Descripción primer problema	
Número de configuraciones	100
Número de cadenas	70

4.2.5 Resumen de pruebas y objetivos

La base de datos dispone entonces de 30 problemas. Sin embargo, debemos considerar lo siguiente. Hay 8 problemas del tipo A para 5 autómatas diferentes. La suma de todas las configuraciones de estos problemas es: 156*2+2+8+58*2+96*2 = 630 . Esto supone que hay 630 instancias diferentes de problemas para el tipo A.

Hay 16 problemas del tipo B para 8 autómatas diferentes. La suma de todas las configuraciones de estos problemas es: 88*2+118*2+54*2+114+122+114*2+88*2+114+114*2+114 = 1616. Esto supone que hay 1616 instancias diferentes de problemas para el tipo B.

Hay 6 problemas del tipo C para 4 autómatas diferentes. La suma de todas las configuraciones de estos problemas es: 25+183*2+22+100*2 = 613 . Esto supone que hay 566 instancias diferentes de problemas para el tipo C.

Esto supone que en total hay 2859 instancias de problemas distintos en la base de datos. Aunque estas instancias se basen en 30 problemas, la forma de resolverlos es muy distinta, puesto que los parámetros están combinados de forma muy variada. Esto supondrá que muchos problemas obtendrán malas soluciones. Sin embargo, el objetivo de las pruebas no es obtener grandes resultados con altos porcentajes de éxito en las soluciones halladas, sino estudiar y analizar los datos para estudiar las limitaciones y ventajas de este tipo de algoritmo, y disponer de un sistema muy probado.

4.2.6 Configuraciones típicas

Hemos tratado de introducir muchas variantes y combinaciones posibles de configuraciones de los parámetros posibles, para tener un amplio espectro de problemas distintos que resolver. Esto nos

permitirá extraer conclusiones de qué configuraciones son más adecuadas para resolver los problemas.

Número de estados

En los problemas en los que hemos considerado un lenguaje (todos los problemas que no están construidos con un autómata aleatorio), hemos construido autómatas finitos mínimos. Y hemos exigido que el número de estados sea igual al número de estados del autómata mínimo. De este modo, aumentamos la dificultad del problema y se lo ponemos difícil al algoritmo.

Población y número de muestras

El valor de población es muy variado en las distintas configuraciones. En algunos problemas comienza desde el valor 100, aumentando hasta 5000, teniendo distintas configuraciones con: 100, 400, 700, 1000, 2000, 3000 y 5000, por ejemplo. En otros problemas más complejos, se omiten los valores de 100 y 400, y se empieza en 500 o 700, puesto que consideramos que probar poblaciones más bajas en esos problemas puede no tener utilidad.

El valor de muestras no varía tanto. Suele tener los valores 20, 40 y en ocasiones 60. Aumentarlo más produce malos resultados. Y tener menos de 20 aumenta las probabilidades de que el algoritmo siempre esté seleccionando a los mismos autómatas para la siguiente generación, impidiendo que vaya mejorando la calidad de los individuos.

Cruzadores, mutadores y calculadores de bondad

Los cruzadores, mutadores y calculadores de bondad van variando. Cada uno de ellos se prueba varias veces con cada población y número de muestras considerado en la lista de configuraciones. Cada uno de estos elementos se combina con muchas permutaciones de los otros. Es decir, prácticamente se combinan todos con todos. Es decir, cada cruzador estará considerado con cada mutador y calculador de bondad en alguna configuración, y lo mismo ocurrirá con los mutadores y los calculadores,

Debemos añadir que en las pruebas no hemos considerado el resolutor de pila, como hemos comentado anteriormente, puesto que el resolutor de vectores es mucho más rápido, y no tiene ninguna desventaja.

4.2.7 Configuraciones especiales

En algunos problemas hemos planteado la posibilidad de ofrecer 1 estado más de los necesarios al algoritmo, para comprobar si esto ayuda al algoritmo a resolver el problema más rápidamente o con más corrección.

En otros problemas, hemos probado otra alternativa más interesante. En la prueba C12 hemos planteado un autómata finito mínimo que tiene 12 estados. Sin embargo, en la configuración hemos pedido que el máximo número de estados sea 10 estados. Igualmente, en la prueba C9, el autómata mínimo tiene 9 estados, pero hemos pedido 7 u 8 estados. Si los resultados son aceptables, es decir que pese a que el porcentaje de reconocimiento no sea del 100%, esté en unos margenes razonables, habremos conseguido obtener un autómata finito que sería imposible de obtener en otro tipo de sistemas, donde sólo se pueden usar AFDs. Al devolver nuestro algoritmo un AFP, puede devolver el mejor autómata posible encontrado para un número de estados para el que es imposible encontrar

un AFD perfecto por ningún otro medio. Esto se podría considerar como una característica muy positiva de nuestro sistema, puesto que puede requerirse en alguna situación un máximo número de estados permitidos.

4.2.8 Resultados esperados

El resultado obtenido por la herramienta para una serie de cadenas de entrada depende de distintos factores. Tuvimos unas primeras impresiones sobre cómo iban a influir los distintos parámetros en la solución, cuando aún no habíamos analizado los resultados. Tras analizar los resultados, algunos de ellos no resultaron como esperábamos. Veamos cuáles eran estos efectos que esperábamos de los parámetros en la ejecución.

Estados

El número de estados es un factor clave en un problema. No podemos esperar buenos resultados si ponemos un número de estados inferior al mínimo número de estados que debe tener un autómata que reconozca o rechace las cadenas de entrada. En este caso, habría un gran número de transiciones con probabilidades alejadas del 0 y del 1. Con mucha suerte, el porcentaje de acierto en el reconocimiento de cadenas podría ser del 60%. Sin embargo, el parecido con un autómata finito, muy probablemente tendrá un valor muy bajo.

Si ponemos un número de estados superior al necesario, esperamos que el algoritmo encuentre un autómata válido como solución al problema. Esto es debido a que si el algoritmo dispone de más estados, entonces las probabilidades de que entre los individuos generados se obtenga un individuo que reconozca las cadenas correctamente, son mayores. Además, la probabilidad de que se parezca a un autómata finito también aumentan, puesto que es posible que exista un autómata finito determinista, no como en el caso anterior. Sin embargo, es posible que, al permitir mayor número de estados, el autómata encontrado no sea el que se había esperado en principio. Pero esto no es un problema, puesto que cumple con el objetivo.

Cadenas de entrada

Longitud

Para cadenas de entrada cortas, que dan lugar a AFD sencillos, la aplicación debería comportarse muy bien, y aportar soluciones iguales o equivalentes a dicho AFD.

Según va aumentado la complejidad o tamaño de las cadenas de entrada, ocurrirán dos cosas. Primero, la calidad de la solución debería descender aunque determinar el largo de las cadenas a partir del cual la calidad empieza a depender puede ser difícil de determinar y dependiente del problema particular. Si tenemos cadenas muy largas, su correcta evaluación involucra a muchas transiciones. Esto supondrá que las transiciones que se irán obteniendo en cada paso según evolucionan los autómatas no serán perfectas (por tanto las probabilidades estarán alejadas del 100% o del 0%), puesto que para intentar reconocer cadenas tan largas, probablemente el algoritmo tenga que repartir las probabilidades entre varios estados diferentes, y por tanto, el autómata obtenido se irá separando del AFD ideal descrito anteriormente.

Segundo, el tiempo de cálculo de la solución aumentará. Cuanto más largas las cadenas de entrada, más se incrementa el tiempo de evaluación de ellas. Aunque sea relativamente pequeño este tiempo,

si la población es de 5.000 individuos por ejemplo, se hace esta comprobación 5.000 veces por iteración. Como el algoritmo tiene un máximo de 50 iteraciones, se puede llegar a evaluar cada cadena 250.000 veces. Si todas las cadenas de entrada son más largas de lo normal, el tiempo de ejecución puede aumentar bastante.

Cantidad

La cantidad de palabras de entrada es otro factor conflictivo. Cuantas más cadenas de entrada haya, más determinado estará el autómata que se quiere obtener, y por tanto, el acierto en el reconocimiento de las cadenas será alto, obteniendo un autómata de calidad.

El inconveniente de un gran número de cadenas de entrada es que el tiempo de ejecución aumentará puesto que hay más cadenas que evaluar para cada autómata.

Por eso, habrá que encontrar un equilibrio: habrá que escoger una cantidad de cadenas suficiente para encontrar un autómata solución de buena calidad pero que esta cantidad no repercutan negativamente en el tiempo de solución innecesariamente.

Población

Para valores bajos de población, el algoritmo genético debería comportarse muy mal. Esto provocaría que los AF obtenidos como resultado de la aplicación tuvieran muchas transiciones cuya probabilidad sea cercana al 50%, es decir que el solapamiento con un AFD sería casi nulo, ya que valores del 50% sería lo que obtendríamos usando únicamente azar puro. Según vayan creciendo los valores de población, la calidad de la solución debería crecer, acercándose cada vez más las probabilidades de las transiciones al 100% (o al 0% para una transición que no debería existir). Con estas probabilidades estaríamos obteniendo autómatas que prácticamente coincidirían con un AFD.

Para valores muy altos de población, el rendimiento de la aplicación se resentirá mucho, tardando demasiado tiempo en aportar una solución, que seguramente no diferirá en gran medida, de una obtenida para unos valores más bajos de población. Sin embargo, un valor alto de población, en consonancia con otros parámetros bien elegidos, nos garantiza que las probabilidades de obtener una buena solución aumenten. Ahí es donde cobra importancia hacer pruebas. Uno de nuestros objetivos con las pruebas, es obtener valores para los parámetros como la población, que guarden un equilibrio entre corrección y rendimiento aunque es razonable esperar que dichos valores dependan del problema particular e incluso de la ejecución particular dadas las características estocásticas del algoritmo.

Muestras

Con las muestras, no se puede determinar a primera vista su comportamiento. Como comentamos anteriormente, en el número de muestras hay que hallar un equilibrio, y no se puede proporcionar un número muy alto ni muy bajo.

Un número alto de muestras parece que garantiza que haya mucha variedad en la población inicial. Sin embargo, entre esta variedad de individuos habrá muchos que no serán muy buenos, y se cruzarán con otros resultando en individuos de poca calidad, perdiendo tiempo en el proceso. Además se disminuye la probabilidad de que los mejores miembros se crucen entre ellos. Un número bajo de muestras evita estos inconvenientes, aunque supone que habrá menos variedad en la población.

Cruzadores, mutadores, calculadores

Los resultados que esperamos con los distintos cruzadores, mutadores y calculadores son en principio similares, puesto que hemos intentado implementarlos utilizando diferentes heurísticas, pero siempre lo más correctas y óptimas posible. Excepto en el caso del calculador de bondad simple, cuyo funcionamiento es bueno; sin embargo, esperamos que su efectividad sea superada por los otros calculadores, puesto que la implementación de los siguientes calculadores es más refinada y tiene en cuenta otros factores.

Sin embargo, esperamos obtener malos resultados con el cruzador y el mutador nulos, puesto que esa es su función, ya que no realizan ningún trabajo. Como hemos comentado anteriormente, éstos sólo existen para probar que los cruzadores y mutadores implementados hacen bien su trabajo, puesto que los nulos ni cruzan ni mutan los individuos.

4.3 Segunda batería de pruebas

4.3.1 Objetivos y motivación de la segunda batería de pruebas

La realización de esta segunda batería de pruebas vino motivada por algunos datos con los que no estábamos satisfechos tras analizar la primera batería de pruebas. Observamos que al ser las configuraciones ligeramente aleatorias, para incluir muchas situaciones diferentes, y muy distintas entre los problemas, podía haber pocas soluciones con determinadas combinaciones de parámetros del algoritmo genético, y sin embargo, podía haber muchas soluciones con otras combinaciones. Por tanto, estos resultados podían tener cierto error. Posiblemente, no tendrían mucho efecto, pero queríamos obtener datos con más exactitud y precisión, para analizar nuestro sistema de la forma más fiable posible.

Los objetivos de esta batería de pruebas son obtener nuevos datos, para menos problemas, pero exigiendo que las configuraciones utilizadas sean exactamente las mismas. Estas configuraciones serán combinaciones de los parámetros del algoritmo idénticas para todos los problemas, donde se prueben todas las combinaciones. Sin embargo, si pretendíamos hacer esto, debíamos eliminar ciertos valores que no nos fueran de utilidad de los valores de los parámetros, puesto que si no, el número de combinaciones posibles sería demasiado elevado.

Por ejemplo, si pretendiéramos utilizar todos los cruzadores (5), mutadores (4) , calculadores de bondad (4), proporcionar por ejemplo 7 poblaciones posibles (100, 400, 700, 1000, 2000, 4000, 5000), y tener 5 posibles números de muestras (10,20,30,40,60), tendríamos un total de 5x4x4x7x5 = 2800 configuraciones posibles. Para que estos datos tuvieran fiabilidad suficiente habría que ejecutar cada posible configuración unas 10 veces (28.000 ejecuciones por problema). Y si pretendíamos introducir 30 problemas como en la anterior batería, tendríamos que calcular un total de 840.000 problemas. Aunque con la primera batería conseguimos 40.000 problemas en menos 2 semanas con 3 ordenadores a tiempo parcial, y con más ordenadores a tiempo completo podríamos conseguir ese número de problemas resueltos en pocas semanas, preferíamos simplificar el sistema, reducir el número de configuraciones lo más posible y reducir el espectro de problemas. Lo hicimos de tal modo que el análisis de las pruebas no perdiera calidad ni rigurosidad y que las soluciones tuvieran una variedad notable de configuraciones.

Para hacer todo esto, también fue necesaria una ligera modificación en el código del servidor, donde se gestiona qué problema enviar al cliente. Anteriormente, se enviaba el problema que menos soluciones tenía, pidiéndola que lo resolviera con una de las configuraciones que tenía ese

problema, pero escogida de forma aleatoria. Éste era otro factor que desviaba ligeramente los resultados puesto que podía haber más soluciones para una configuración que para otra, y obtener resultados algo variables en casos concretos. Por ello, modificamos esta parte del código, escogiendo siempre, además del problema, la configuración que tuviera menor número de soluciones en la base de datos. Así, siempre se mantendría el mismo número de soluciones para cada configuración.

De este modo, obtendremos datos altamente fiables, puesto que habrá exactamente el mismo número de soluciones para cada configuración. Y como todos los problemas tienen las mismas configuraciones, se podrán extraer datos generales con mucha precisión.

4.3.2 Características de cada problema

Entre las características de cada problema encontramos:

- Tendrá asociados un número fijo de configuraciones para cada problema. En cada una, variará al menos algún parámetro del algoritmo, y se combinarán todos los parámetros de todas las formas posibles, acotando el dominio de los parámetros como se describe a continuación.

  Se utilizarán los 5 cruzadores, 3 mutadores (mutador nulo, tipo 1 y tipo 3), y 2 calculadores de bondad (el balanceado y de preferencia determinista). Se utilizarán como posibles valores de población: 500, 2000 y 4000. Y como valores posibles de muestras: 10, 20 y 40. Esto hace un total de 5x3x2x3x3 = 270 combinaciones posibles, lo que supone que exijamos 270 configuraciones para cada problema.

- Disponemos de una base de datos de 11 problemas. Al disponer de menos problemas que en la otra ocasión, podremos obtener más soluciones en menos tiempo. Para obtener unos resultados muy fiables y ser algo exigentes, necesitamos al menos 10 soluciones por configuración (es decir, 2700 soluciones por problema), haciendo un total de 29.700 soluciones necesarias. Es decir, nos gustaría superar las 30.000 soluciones para esta batería de pruebas.

4.3.3 Notación

Se respetará la misma notación que en la primera batería de pruebas. Por tanto, habrá problemas de tipo A (básicas), tipo B (intermedias) y tipo C (avanzadas). Además, varias de las pruebas realizadas en esta batería de pruebas, coincidirán con problemas de la primera batería, aunque cambiarán muchos parámetros de las pruebas (la más importante, las configuraciones, también las cadenas, y en ocasiones el número de estados). Se introducirá una nueva prueba: B9.

4.3.4 Detalle de las pruebas

Pruebas básicas (A)

Las dos pruebas que consideramos en esta segunda batería coinciden con dos problemas de las pruebas básicas de la primera batería, aunque varían sus configuraciones y sus cadenas:

Nombre de la prueba	Prueba A1

Lenguaje que define el autómata	Lenguaje que acepta las palabras que contienen algún 1.
Número de estados del autómata	2 estados
Número de problemas con el lenguaje	1 problema
Descripción	
Número de configuraciones	270
Número de cadenas	30

Nombre de la prueba	Prueba A3
Lenguaje que define el autómata	Lenguaje que acepta las palabras que contienen la secuencia 01.
Número de estados del autómata	3 estados
Número de problemas con el lenguaje	1 problema
Descripción	
Número de configuraciones	270
Número de cadenas	15

Pruebas intermedias (B)

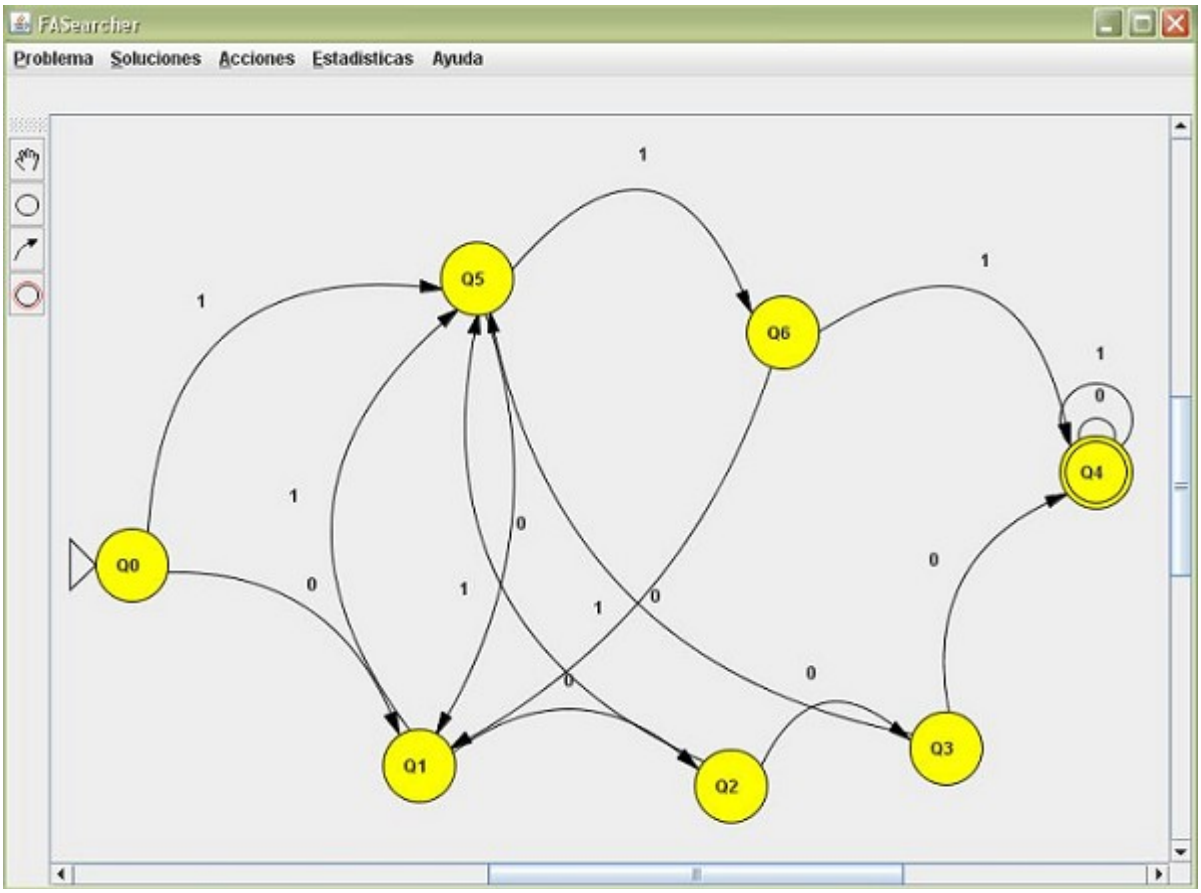
Las pruebas B1, B2 y B3 comparten el mismo autómata que las pruebas respectivas en la primera batería de pruebas. Éstas exigirán el número de estados mínimo para ese lenguaje. La prueba B9 será nueva, y exigirá en una de sus versiones 1 estado menos de los necesarios para el autómata mínimo, y 2 estados menos de los necesarios en la otra versión.

Nombre de la prueba	Prueba B1
Lenguaje que define el autómata	Lenguaje que acepta las palabras que contengan la secuencia 1001.
Número de estados del autómata	5 estados
Número de problemas con el lenguaje	1 problema
Descripción primer problema	
Número de configuraciones	270
Número de cadenas	30

Nombre de la prueba	Prueba B2
Lenguaje que define el autómata	Lenguaje que acepta las palabras que contengan 5 o más ceros.
Número de estados del autómata	6 estados
Número de problemas con el lenguaje	1 problema
Descripción primer problema	
Número de configuraciones	270
Número de cadenas	24

Nombre de la prueba	Prueba B3
Lenguaje que define el autómata	Lenguaje que acepta las palabras que contengan las secuencias 100 o 011.
Número de estados del autómata	6 estados
Número de problemas con el lenguaje	2 problema
Descripción primer problema	
Número de configuraciones	270
Número de cadenas	20
Descripción segundo problema	
Número de configuraciones	270
Número de cadenas	20

Nombre de la prueba	Prueba B9
Lenguaje que define el autómata	Lenguaje que acepta las palabras que contengan 4 ceros o 3 unos consecutivos.
Número de estados del autómata	7 estados
Número de problemas con el lenguaje	2 problema
Descripción primer problema	
Número de estados exigido	5
Número de configuraciones	270
Número de cadenas	20
Descripción segundo problema	
Número de estados exigido	6
Número de configuraciones	270
Número de cadenas	20



Pruebas avanzadas (C)

Hay dos pruebas de este tipo en esta batería de pruebas, que coinciden con las que pertenecían a la primera batería de pruebas. Al igual que en la otra batería, para C12, cuyo autómata mínimo que representa su lenguaje tiene 12 estados, se le exige que busque un autómata con 10 estados, por debajo del mínimo.

Nombre de la prueba	Prueba C9
Lenguaje que define el autómata	Lenguaje que acepta: cadenas que empiecen por 0 y contengan la secuencia 101, o cadenas que empiecen por 1 y tengan 4 unos consecutivos.
Número de estados del autómata	9 estados (mínimo para ese lenguaje)
Número de problemas con el lenguaje	1 problema
Descripción primer problema	
Número de configuraciones	270
Número de cadenas	33

Nombre de la prueba	Prueba C12
Lenguaje que define el autómata	Cadenas que contienen al menos 6 ceros o 6 unos consecutivos
Número de estados del autómata	12 estados (autómata mínimo para ese lenguaje)
Número de problemas con el lenguaje	1 problema
Descripción primer problema	
Número de estados exigido	10
Número de configuraciones	270
Número de cadenas	70

5 Resultados

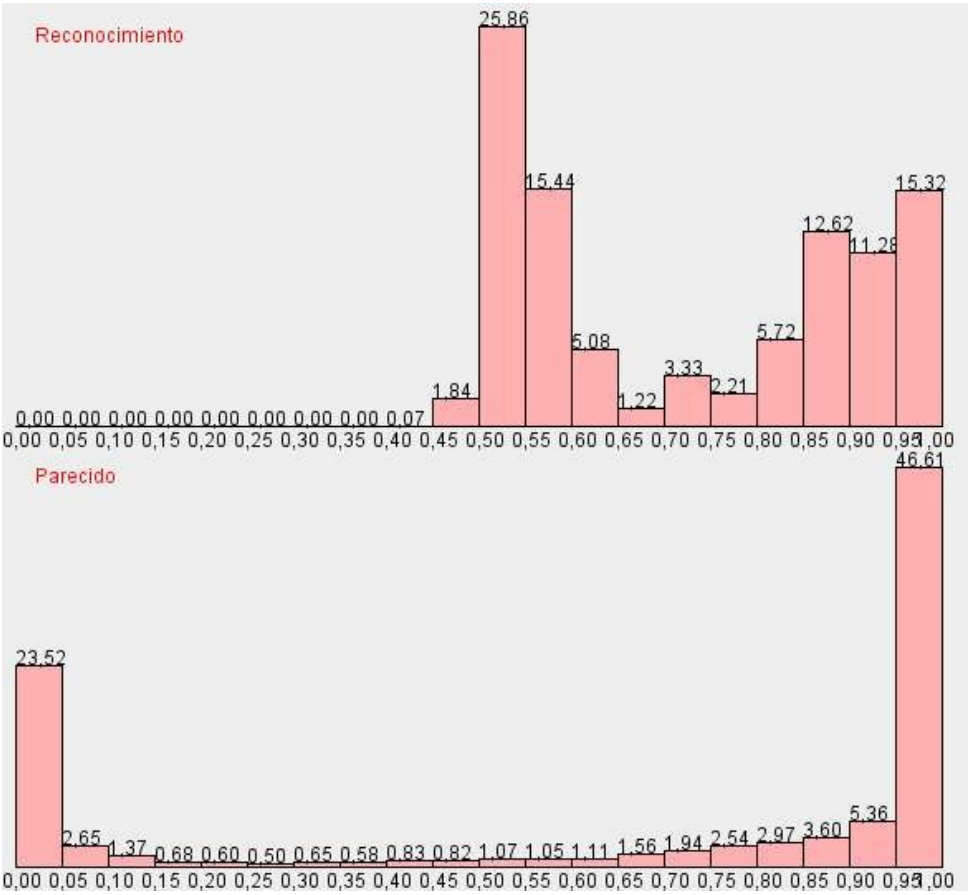
A continuación pasamos a mostrar de forma ordenada, analizar y comparar los resultados de las pruebas. Podemos decir que los resultados mostrados a continuación están respaldados por las más de 70.000 soluciones a problemas que tenemos en la base de datos (más de 40.000 soluciones para la primera batería de pruebas y unas 35.000 soluciones para la segunda batería de pruebas), lo que permite realizar estudios estadísticos de relevancia.

Nos centraremos más en los resultados de la segunda batería de pruebas, puesto que como hemos comentado anteriormente, nuestro objetivo con la segunda batería de pruebas era obtener resultados muy fiables, que fueran obtenido de un conjunto de configuraciones muy homogéneo e idéntico para todos los problemas.

5.1 Resultados de la segunda batería de pruebas

5.1.1 Resultados generales

En esta batería de pruebas, teníamos 11 problemas. Como hemos comentado anteriormente, pretendíamos tener por lo menos 10 soluciones por configuración. Al tener 270 soluciones necesitábamos unas 29.000 soluciones. Finalmente, hemos obtenido cerca de 36.000, más que suficientes para obtener resultados de una fiabilidad notable. A continuación mostramos los resultados generales de los 11 problemas, para las 36.000 soluciones.



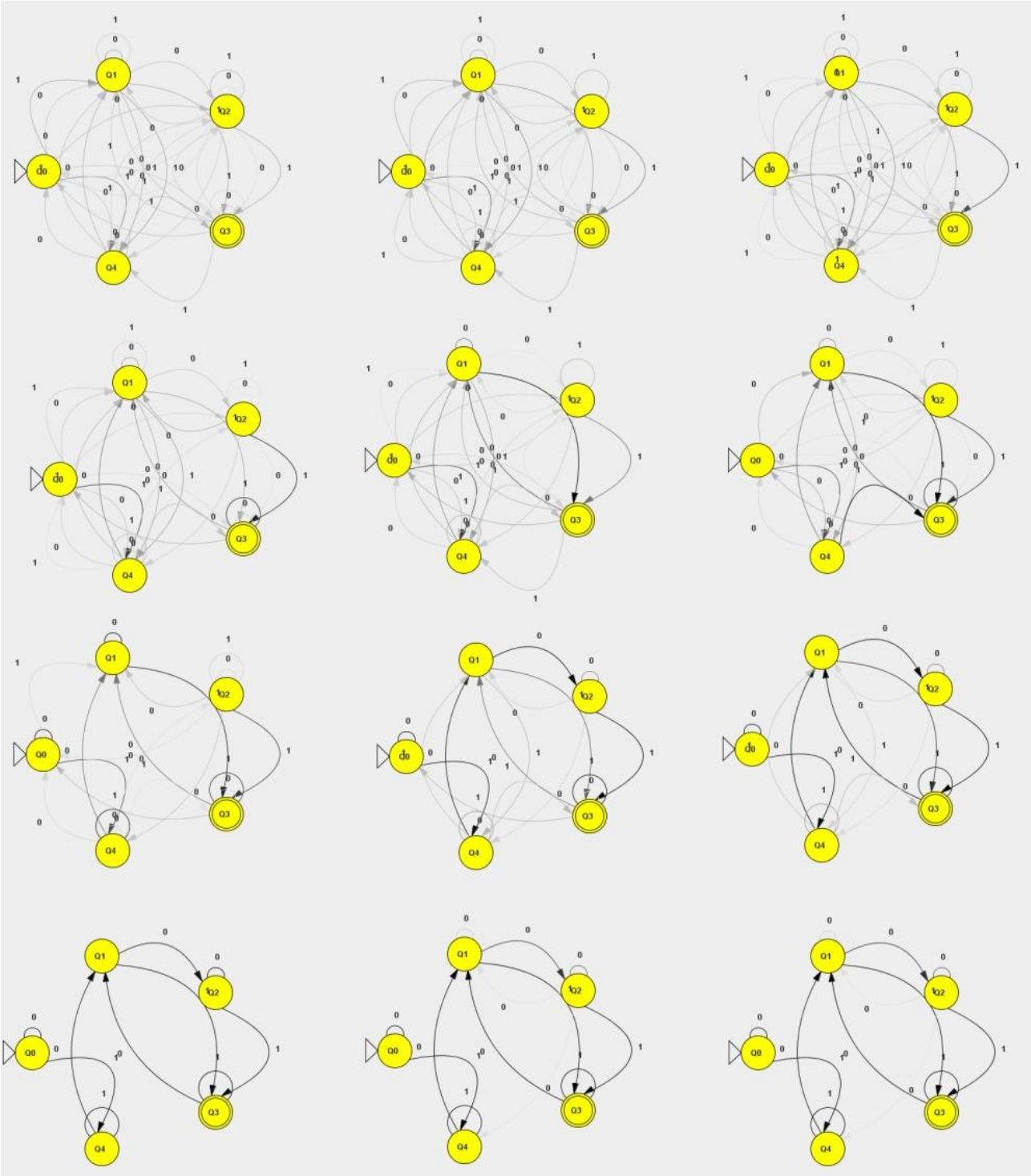
Si consideramos buenos resultados aquellos superiores al 90% en acierto al reconocer cadenas, observamos que el 26% (11,28+15,32) de las soluciones cumplen este requisito. Este porcentaje es bajo. Sin embargo, debemos tener en cuenta que son los resultados para las soluciones de todas las configuraciones posibles.

Y como ya se ha comentado en la descripción de las pruebas, en éstas se incluyen lo que podríamos llamar "casos desfavorables", esto es, configuraciones que no tienen que dar buenos resultados sino que están destinadas a probar el algoritmo en distintos casos para extraer conclusiones. Muchas de las configuraciones serán desfavorables, puesto que se incluyen los cruzadores y mutadores nulos, bajas poblaciones y distintos calculadores de bondad. A continuación, extraeremos unos datos que nos permitirán filtrar los mejores parámetros para las configuraciones.

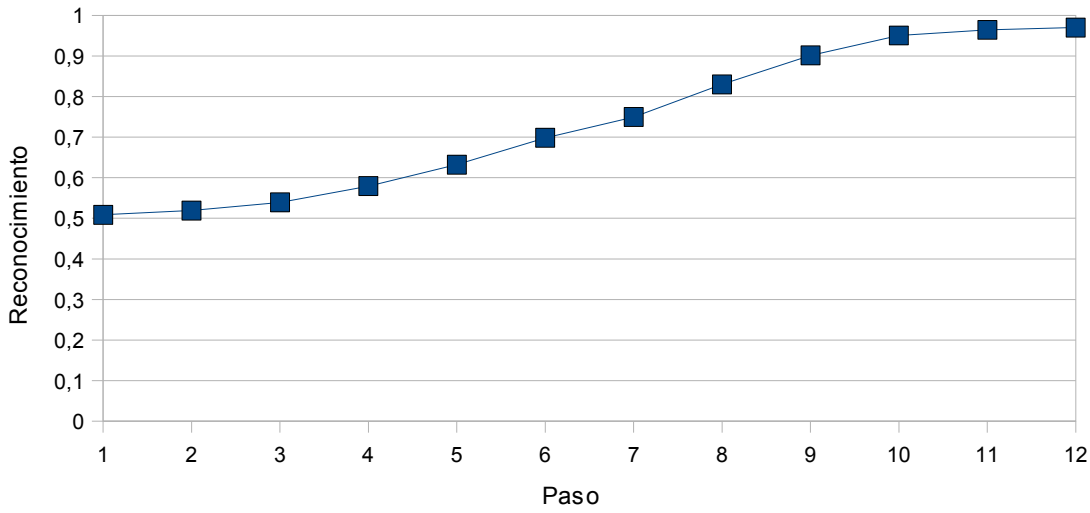
Como resultado de esto, los resultados generales no son representativos de cómo funciona el algoritmo sino que son representativos del contenido de la base de datos.

Ejemplo de como mejoran los autómatas en cada iteración

A continuación, se puede ver con un ejemplo como es el mejor AFP seleccionado en cada una de las iteraciones del algoritmo. Esta información también se puede ver con el cliente y resulta muy interesante ver los cambios que se van realizando:



Además, podemos ver como en este ejemplo aumenta el porcentaje de reconocimiento del lenguaje en cada paso:

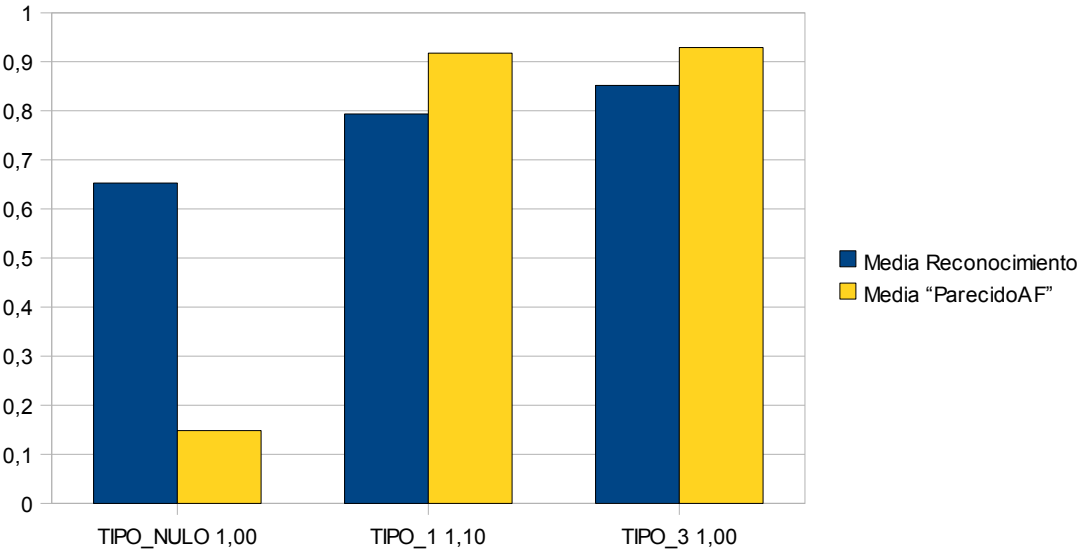


Este gráfico será distinto según el problema y la configuración, pero en general se repite la forma de éste y las características fundamentales: comienza en valores en torno a 0,5 , los valores mejoran con el número de iteraciones y se acerca de forma más lenta a 1 en las últimas iteraciones (en muchos casos no alcanzando nunca el valor de 1 exacto).

5.1.2 Estudio de los elementos

Comparación entre los mutadores

En la gráfica que se muestra a continuación comparamos como se comportan cada uno de los 3 mutadores utilizados en el algoritmo genético. Para ello usamos dos valores que nos indican como se ha comportado el algoritmo con cada uno de los mutadores, en las áreas que nos interesa probar su eficiencia: el reconocimiento de las cadenas propuestas como entrada y el parecido a un AF. Los datos están tomados de la base de datos consultando la media de los valores medidos para todas las soluciones de todos los problemas, agrupando los resultados por mutadores.



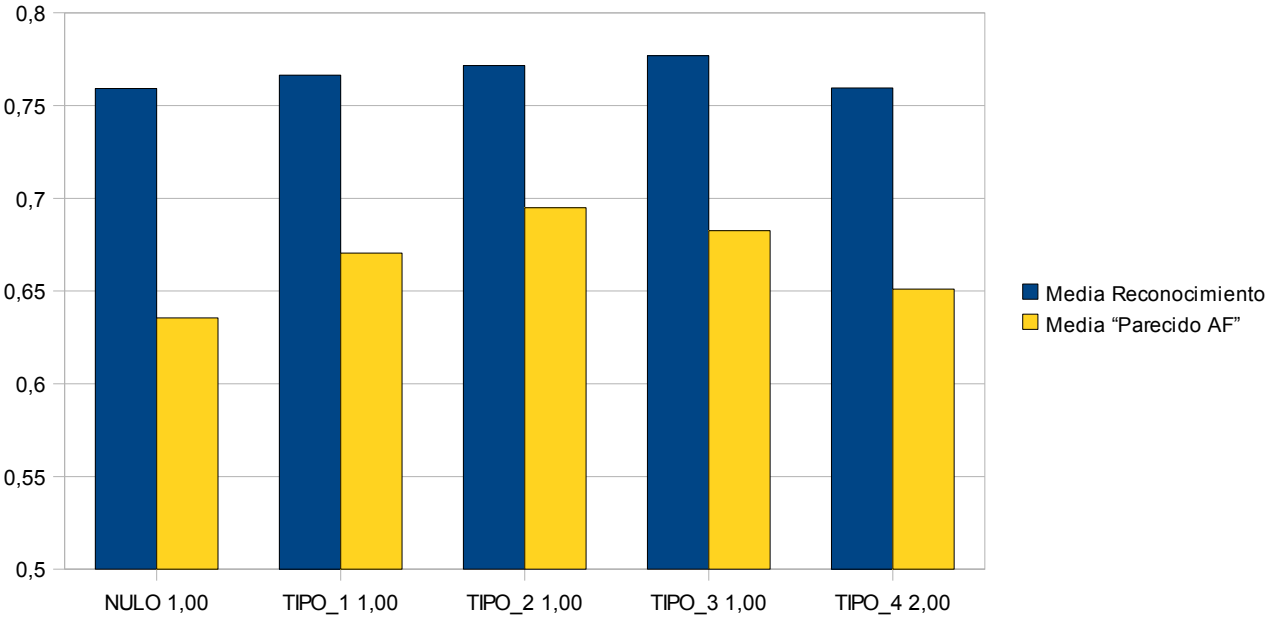
Como se puede apreciar, es muy importante que el algoritmo cuente con un mutador. El mutador

nulo provoca que el parecido final con un AF sea casi inexistente, y la tasa de reconocimiento es claramente inferior a la obtenida usando alguno de los otros dos mutadores.

Respecto a los dos mutadores no nulos, ambos tienen un comportamiento semejante, con unas buenas tasas tanto de reconocimiento como de parecido a un AF. Pese a ello, se puede ver como el mutador de tipo 3 es ligeramente superior al de tipo 1.

Comparación entre los cruzadores

Como en el caso anterior, vamos a utilizar los valores de reconocimiento de las cadenas de entrada y el parecido a un AF, para comparar el comportamiento de los cruzadores del algoritmo genético. Al igual que en el caso anterior, se ha consultado la base de datos, seleccionando la media de los valores de todas las soluciones de todos los problemas, agrupando por cruzadores.



Al contrario que en el caso anterior, la ausencia de cruzador no tiene tanta influencia, y sus valores son bastante semejantes a los que se obtienen usando los otros cruzadores. De hecho, pese a que el valor de parecido a un AF es el mínimo respecto al resto de casos, el reconocimiento de las cadenas de entrada, es ligeramente superior al obtenido por el mutador de tipo 4.

El comportamiento de los cruzadores no nulos es también muy semejante entre sí. Los valores tanto de reconocimiento como de parecido, son muy homogéneos, sin que ninguno sea capaz de sobresalir en los dos campos. El cruzador de tipo 3 es el que obtiene el mejor valor para el reconocimiento de cadenas de entrada, y el de tipo 2 para el parecido a un AF.

De todo esto podemos deducir que la importancia del cruzador en nuestro algoritmo es menor de lo que se podía pensar en un principio. Todos los cruzadores, incluso el nulo, se comportan de una manera muy similar, y dependerá de cada uno de los problemas particulares, que uno de ellos pueda sobresalir ligeramente sobre los otros.

Conclusiones sobre los mutadores y cruzadores

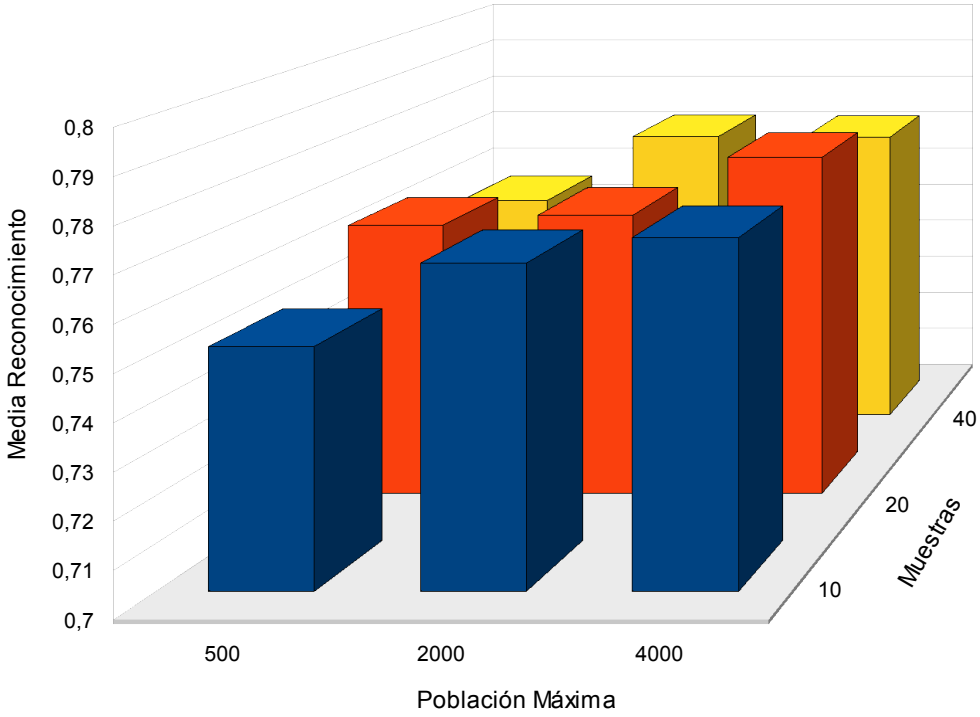
Al agrupar los datos de los mutadores por un lado, y los cruzadores por otro, extraemos unas

conclusiones interesantes. En el primer caso, cogemos para cada mutador todas las soluciones con todas las combinaciones de los otros parámetros. Y en el segundo caso, cogemos para cada cruzador, todas las soluciones de combinaciones de los otros parámetros. Al observar grandes diferencias entre los distintos mutadores, y ver que la media de reconocimiento es más alta que al agrupar por cruzadores extraemos el resultado de que el mutador es la pieza más importante del algoritmo, puesto que independientemente del cruzador u otros elementos, el mutador 3 obtiene grandes resultados. Sin embargo, al observar los cruzadores, vemos que los resultados apenas varían entre unos y otros. Esto ocurre porque los cambios importantes se producen al cambiar otros parámetros, y las medias de reconocimiento de cadenas utilizando un cruzador u otro resultan muy semejantes por este motivo.

Por este motivo, al agrupar los problemas por mutador, observamos que la media de reconocimiento obtenida por el mutador TIPO_3 supera el 85%. Sin embargo, las medias de cualquiera de los cruzadores no alcanza el 78%.

Análisis de la población máxima y el número de muestras

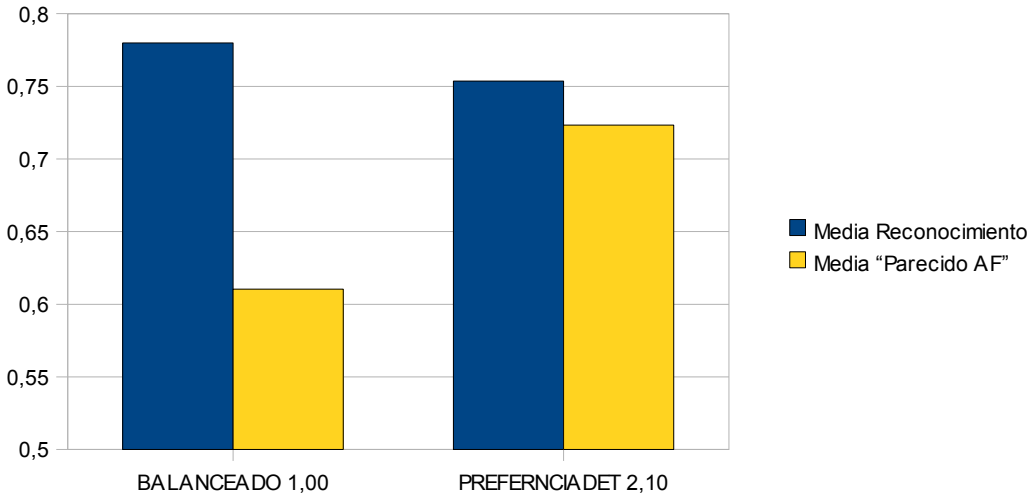
Para el análisis de la importancia de la población máxima y el número de muestras, usamos como valor de referencia la media de reconocimiento de cadenas de entrada obtenida. En la gráfica, colocamos la población máxima en el eje X y el número de muestras en el Z.



Al igual que ha ocurrido en el caso de los cruzadores, la importancia final del número de muestras y la población máxima ha sido menor de la esperada en un principio. Los resultados obtenidos son prácticamente idénticos en todos los casos. Como se puede apreciar en la gráfica, las variaciones se producen en un margen inferior a una décima (el eje Y oscila entre 0,7 y 0,8). Por lo tanto, con unos valores no extremos de población y muestras (como los que se muestran en la gráfica), los resultados que se obtienen son buenos, y no hay apenas diferencia entre la calidad de unos u otros.

Comparación entre los calculadores de bondad

Al igual que hicimos en el caso de los mutadores y los cruzadores, volvemos a usar los valores de reconocimiento de cadenas de entrada y el parecido a un AF, para comparar los dos calculadores de bondad que hemos utilizado en las pruebas.



En el caso del calculador Balanceado puede verse cómo se obtiene una gran tasa de reconocimiento, pero a cambio, se pierde parecido con AF. Esto es debido a que este calculador no obtiene

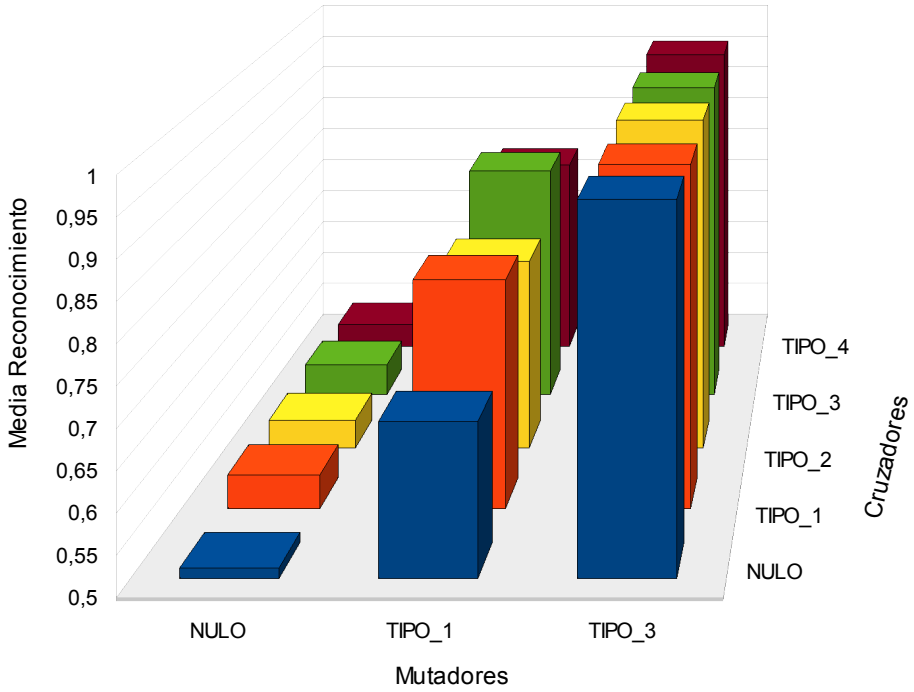
El de Preferencia Determinista (PreferenciaDet) sin embargo, debido a que en su implementación considera una parte importante de la bondad de un individuo su parecido a un AF, obtiene una tasa muy superior en ese campo respecto al otro calculador. Por contrapartida, el valor de reconocimiento, baja ligeramente, tan sólo unas 2 centésimas (puesto que al dar importancia al parecido con un AF, puede perder ligeramente la precisión del calculador balanceado al encontrar un AFP de gran calidad en el reconocimiento de palabras).

En cualquier caso, los dos calculadores obtienen unos buenos valores en ambos campos, y buscan cumplir funciones distintas. Por lo tanto, según el problema deseado, será más apropiado usar uno u otro.

5.1.3 Estudio de los problemas por tipo

Problemas de tipo A

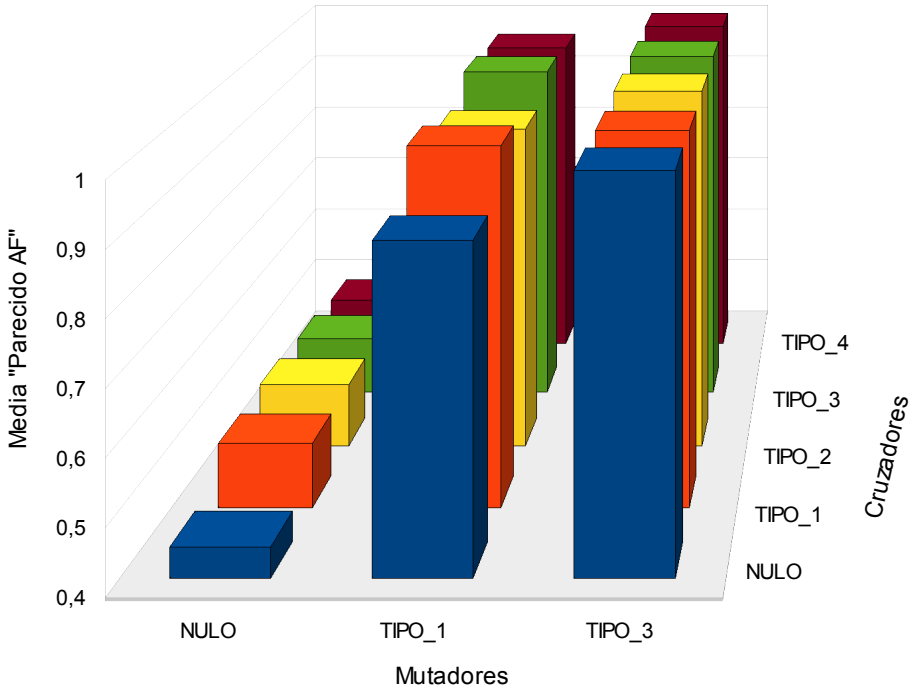
Ahora vamos a analizar los resultados obtenidos en los problemas del tipo A, es decir, pruebas sobre autómatas con pocos estados. Comparamos, en dos gráficas separadas, el reconocimiento y el parecido a AF, para cada tipo de mutador (eje X) y de cruzador (eje Z).



Como puede apreciarse en la gráfica, claramente la mejor opción es usar el mutador tipo 3 para este tipo de problemas. Usando este mutador, la diferencia entre usar un cruzador u otro (o incluso no usarlo) es prácticamente nula, y la tasa de reconocimiento es muy cercana al 1.

También se ve que no utilizar un mutador provoca que los resultados que se obtienen sean muy malos, con un reconocimiento cercano al 0,5, para cualquiera que sea el cruzador utilizado.

Por último, en el caso de mutador de tipo 1, sí hay más variación según el cruzador utilizado. El nulo y el tipo 2, obtienen las tasas más bajas, siendo el de tipo 3 el más recomendable, ya que es el que mejores valores obtiene.



Como ocurre en el caso anterior, el mutador de tipo 3 es la mejor opción, pero con poca diferencia respecto al mutador tipo 1. Al igual que para el reconocimiento, el tipo 3 obtiene un parecido con AF cercano al 100%, para todos los posibles cruzadores.

El mutador tipo 1 aumenta su calidad respecto a la calidad vista en la media de reconocimiento. Los cruzadores de nuevo, tienen un efecto poco significativo. Las peores opciones siguen siendo el cruzador nulo y el de tipo 2.

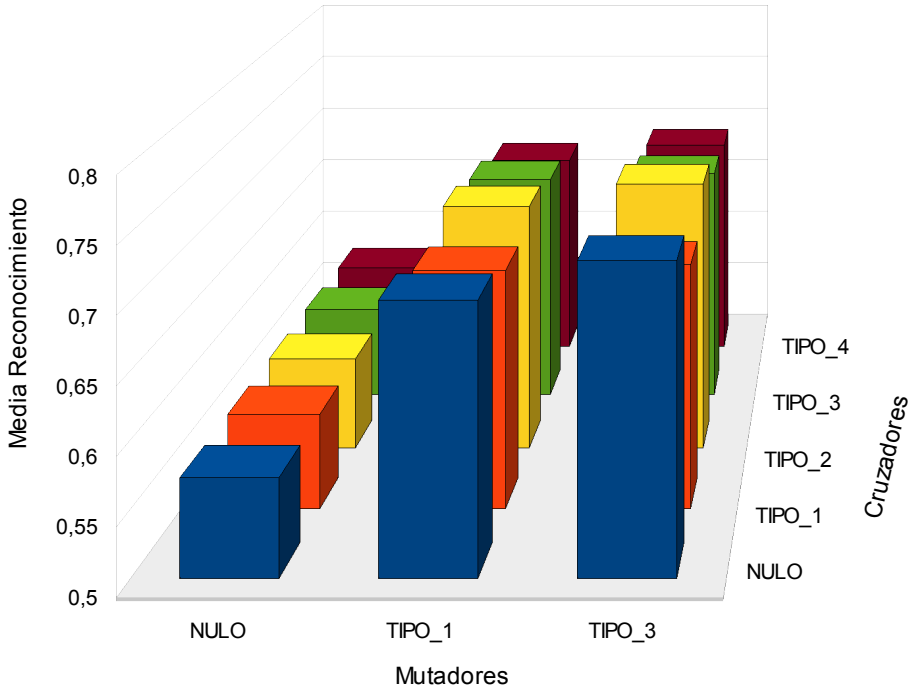
Finalmente, al igual que en el caso anterior, el no usar mutador provoca que los resultados obtenidos sean muy bajos.

Por último, en esta tabla mostramos los mejores valores obtenidos en esta batería de pruebas. Puede verse como para casi todos estos casos, en 24 ejecuciones se ha obtenido unas tasas de reconocimiento y parecido a AF de 1, es decir, un 100%.

Media Reconocimiento	Varianza Reconocimiento	Media "Parecido AF"	Pasos	Número de soluciones	Cruzador	Mutador	Muestras	Población Máxima	Calculador Bondad
0,99162	0,00160	0,99996	4	24	TIPO_31,00	TIPO_3 1,00	20	4000	BALANCEADO 1,00
0,99490	0,00060	0,99707	8	24	TIPO_31,00	TIPO_3 1,00	20	2000	BALANCEADO 1,00
1,00000	0,00000	1,00000	4	24	NULO 1,00	TIPO_3 1,00	10	4000	BALANCEADO 1,00
1,00000	0,00000	1,00000	3	24	NULO 1,00	TIPO_3 1,00	40	2000	BALANCEADO 1,00
1,00000	0,00000	1,00000	3	24	TIPO_42,00	TIPO_3 1,00	10	4000	PREFERNCIADET 2,10
1,00000	0,00000	1,00000	3	24	TIPO_2 1,00	TIPO_3 1,00	20	4000	BALANCEADO 1,00

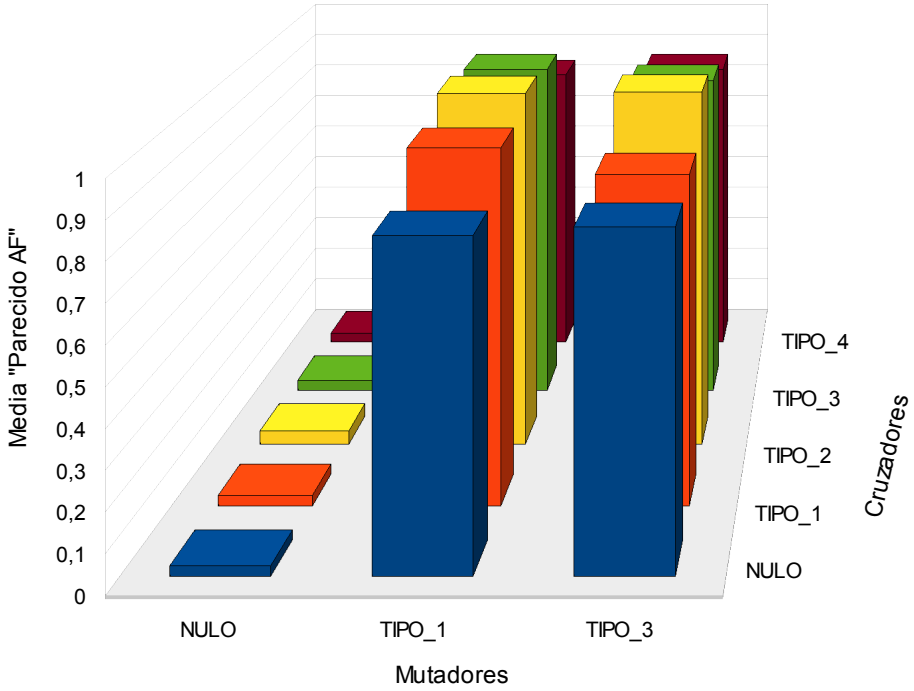
Problemas de tipo B

Pasamos a estudiar los resultados de los problemas del tipo B. Estos problemas, sobre autómatas de tamaño medio, buscan obtener autómatas con un número de estados mínimo, o incluso inferior a éste. Al igual que en las pruebas de tipo A, usamos el mismo formato de gráficas en 3D y por último una tabla con los mejores valores obtenidos.



Como en todos los casos anteriores, la ausencia de mutador sigue provocando los peores valores sea cual sea el cruzador utilizado.

En este caso, la diferencia entre el mutador de tipo 1 y el de tipo 3 es mínima. Sólo difieren en el caso de no usar cruzador y en el de usar el de tipo 2, pero incluso en estos casos, la diferencia a favor del mutador de tipo 3 es muy pequeña.



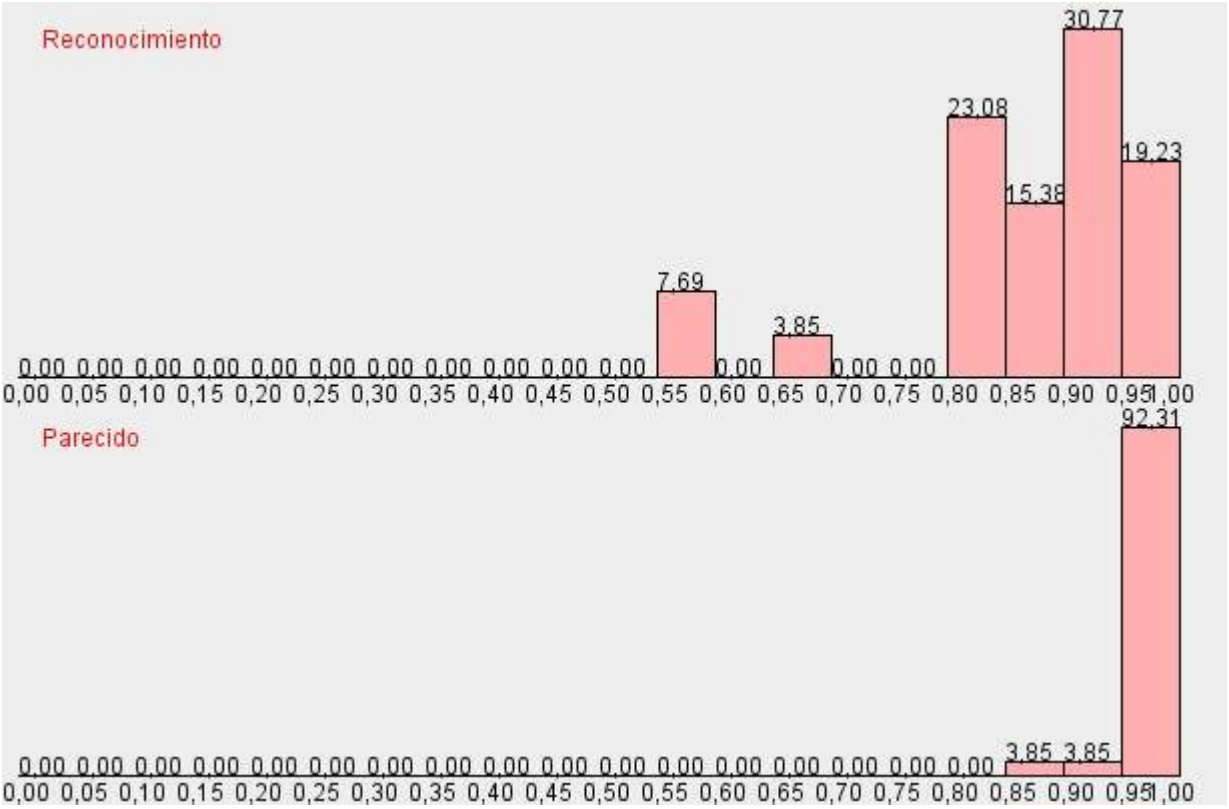
En esta gráfica de nuevo se aprecia que los valores obtenidos con el mutador de tipo 1 y del tipo 3 son muy semejantes, y que el uso del mutador nulo, sigue provocando valores mínimos, cada vez más cercanos al 0.

Los valores de parecido a un AF suben respecto al de reconocimiento para los mutadores de tipo 1 y 3, acercándose en algunos casos al 0,9. En este caso, también se puede apreciar, que el mutador de tipo 1 es ligeramente superior al de tipo 3 usando los cruzadores de tipo 1 y 3.

Por último, la gráfica ilustra lo obtenido en las gráficas. Los valores de reconocimiento, siendo estos los mejores obtenidos, oscilan entre el 0,8 y el 0,85 y los de parecido a AF son superiores, acercándose mucho al 1, pero sin llegar a él.

Media Reconocimiento	Varianza Reconocimiento	Media "Parecido AF"	Pasos	Número de soluciones	Cruzador	Mutador	Muestras	Población Máxima	Calculador Bondad
0,80045	0,02639	0,94543	13	24	TIPO_2 1,00	TIPO_1 1,10	40	2000	BALANACEADO 1,00
0,80296	0,02849	0,96613	16	24	TIPO_2 1,00	TIPO_3 1,00	20	500	BALANACEADO 1,00
0,80885	0,01385	0,82214	50	23	TIPO_31,00	TIPO_3 1,00	40	500	BALANACEADO 1,00
0,82214	0,03190	0,86664	50	24	TIPO_31,00	TIPO_3 1,00	20	500	BALANACEADO 1,00
0,82502	0,02804	0,93880	50	23	TIPO_31,00	TIPO_1 1,10	40	2000	BALANACEADO 1,00
0,83282	0,02619	0,80505	9	24	NULO 1,00	TIPO_3 1,00	20	500	BALANACEADO 1,00
0,83578	0,01745	1,00000	13	24	NULO 1,00	TIPO_3 1,00	20	500	PREFERNCIADET 2,10
0,83928	0,01384	0,85765	18	24	TIPO_1 1,00	TIPO_3 1,00	40	2000	BALANACEADO 1,00
0,83960	0,01632	0,87965	18	24	NULO 1,00	TIPO_3 1,00	40	500	BALANACEADO 1,00

En este caso puede ser interesante ver como se distribuyen las soluciones del problema para el mejor tipo de configuración:

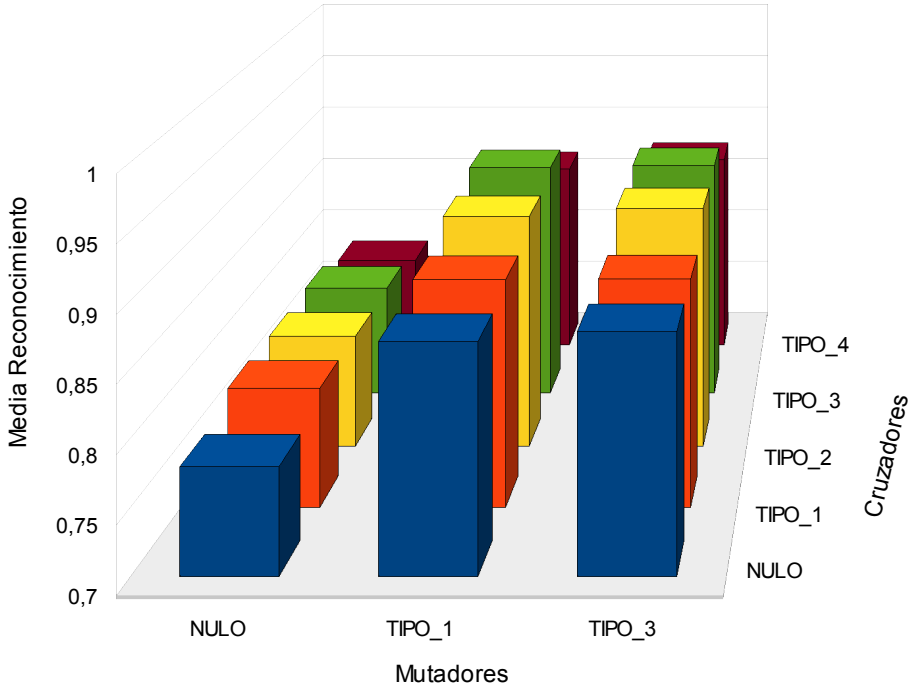


Y podemos ver que, aunque no todas las soluciones son perfectas, hay una clara tendencia a obtener un reconocimiento mayor al 80%.

Problemas de tipo C

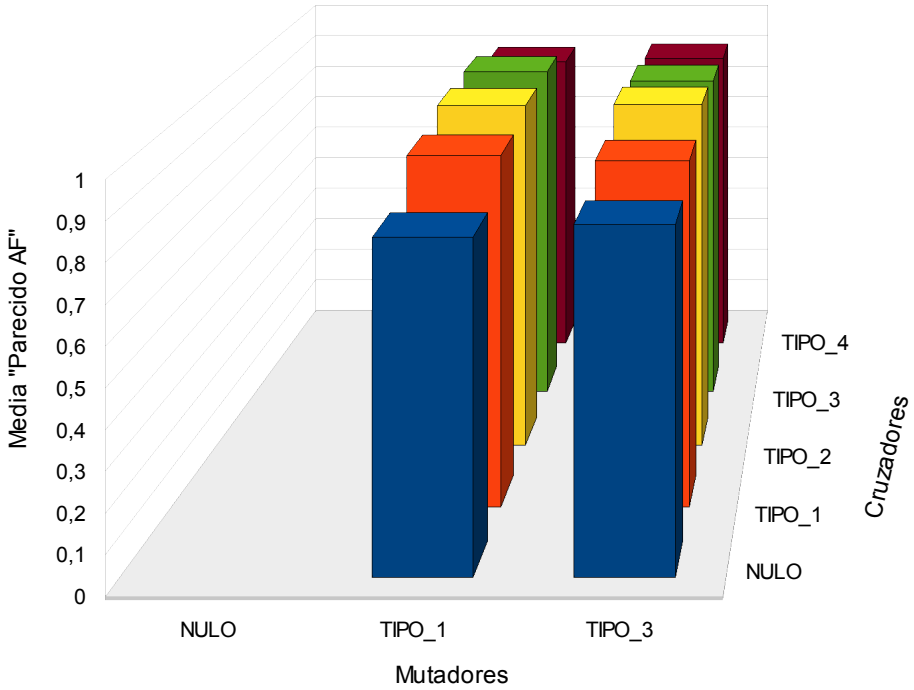
Por último analizaremos las pruebas de tipo C. De nuevo se buscan autómatas mínimos o con un

número de estados menor a éste. En este caso, los autómatas utilizados son grandes, con un gran número de estados.



Éste es el caso en el que el mutador nulo ofrece el mejor comportamiento, con una tasa de reconocimiento para todos los cruzadores del 0,75.

De nuevo, los mutadores de tipo 1 y 3 se comportan de una manera muy similar, al igual que todos los cruzadores. En general, ambos mutadores obtienen unas tasas de reconocimiento, que según el tipo de cruzador oscilan entre el 0,85 y el 0,95.



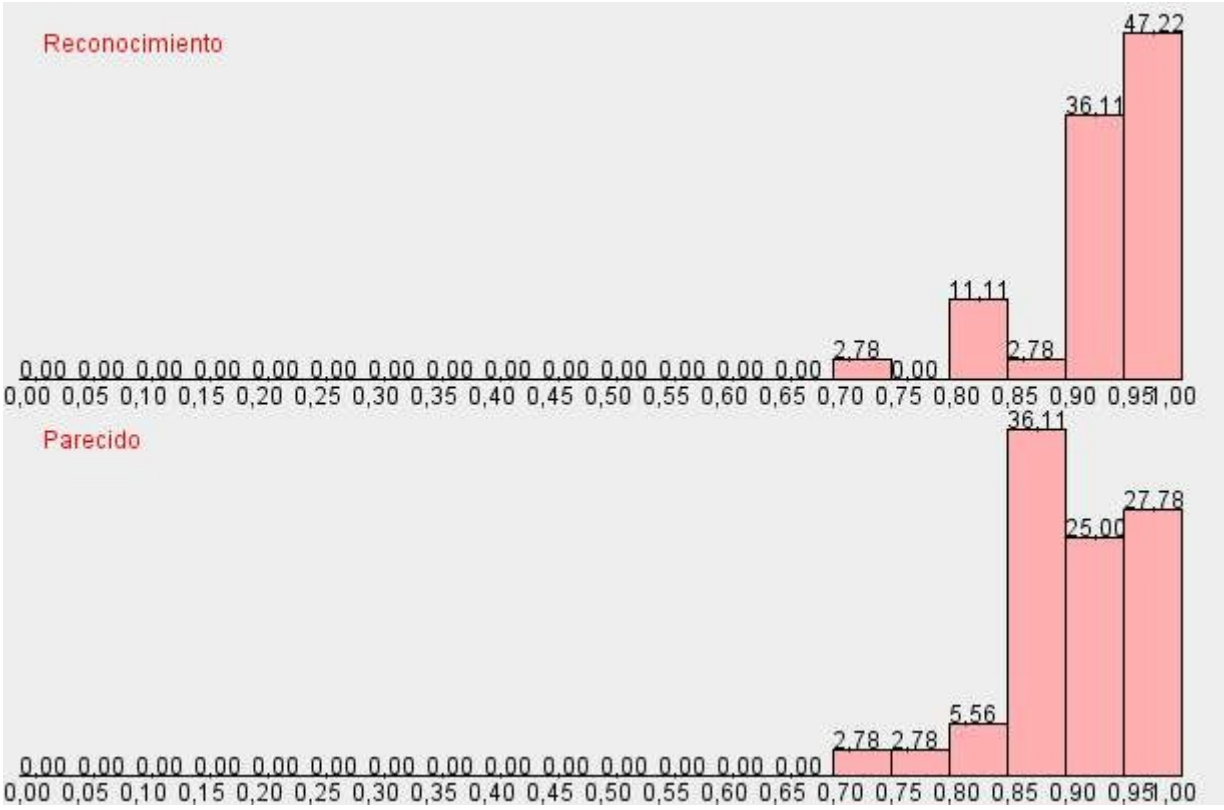
Como contrapartida al caso del reconocimiento, dónde el mutador nulo obtenía sus mejores resultados en todas las pruebas analizadas, para el parecido a AF, el mutador nulo obtiene valores de 0 para todos los cruzadores. Esto hace inviable su utilización para este tipos de pruebas.

Como en el caso anterior, los mutadores de tipo 1 y 3 se comportan de manera muy similar, así como los distintos tipos de cruzadores, obteniendo valores de parecido con AF muy altos, superiores al 0,8. Sólo el cruzador nulo obtiene unos valores ligeramente inferiores al resto, pero en cualquier caso, éstos siguen dentro de un rango aceptable.

Entre los mejores resultados, los valores de reconocimiento oscilan todos en un 0,9 bajo, como se pudo apreciar anteriormente, y se muestra en la siguiente tabla. Los de parecido a AF, ofrecen un mayor rango de variación, empezando en valores cercanos al 0,9 y llegando incluso al 1.

Media Reconocimiento	Varianza Reconocimiento	Media "Parecido AF"	Pasos	Número de soluciones	Cruzador	Mutador	Muestras	Población Máxima	Calculador Bondad
0,91387	0,00272	1,00000	11	35	TIPO_2 1,00	TIPO_1 1,10	10	2000	PREFERNCIADET 2,10
0,91456	0,00399	0,86726	29	36	TIPO_31,00	TIPO_3 1,00	10	500	BALANACEADO 1,00
0,91473	0,00332	0,88340	50	34	TIPO_31,00	TIPO_3 1,00	10	2000	BALANACEADO 1,00
0,91551	0,00380	0,92205	29	35	TIPO_2 1,00	TIPO_1 1,10	40	4000	BALANACEADO 1,00
0,91641	0,00290	1,00000	12	36	NULO 1,00	TIPO_3 1,00	10	2000	PREFERNCIADET 2,10
0,91651	0,00174	0,99643	33	35	TIPO_31,00	TIPO_3 1,00	20	2000	PREFERNCIADET 2,10
0,91681	0,00297	0,94845	23	36	TIPO_2 1,00	TIPO_3 1,00	20	2000	BALANACEADO 1,00
0,91760	0,00254	0,92565	41	36	TIPO_31,00	TIPO_1 1,10	10	2000	BALANACEADO 1,00
0,91863	0,00235	0,88272	50	35	TIPO_31,00	TIPO_1 1,10	40	2000	BALANACEADO 1,00
0,92208	0,00178	0,92671	50	35	TIPO_2 1,00	TIPO_1 1,10	10	500	BALANACEADO 1,00
0,92259	0,00294	0,86938	50	36	TIPO_31,00	TIPO_3 1,00	20	2000	BALANACEADO 1,00
0,92529	0,00168	0,90288	50	35	TIPO_31,00	TIPO_1 1,10	10	500	BALANACEADO 1,00
0,92667	0,00221	0,94267	26	35	TIPO_2 1,00	TIPO_3 1,00	40	4000	BALANACEADO 1,00

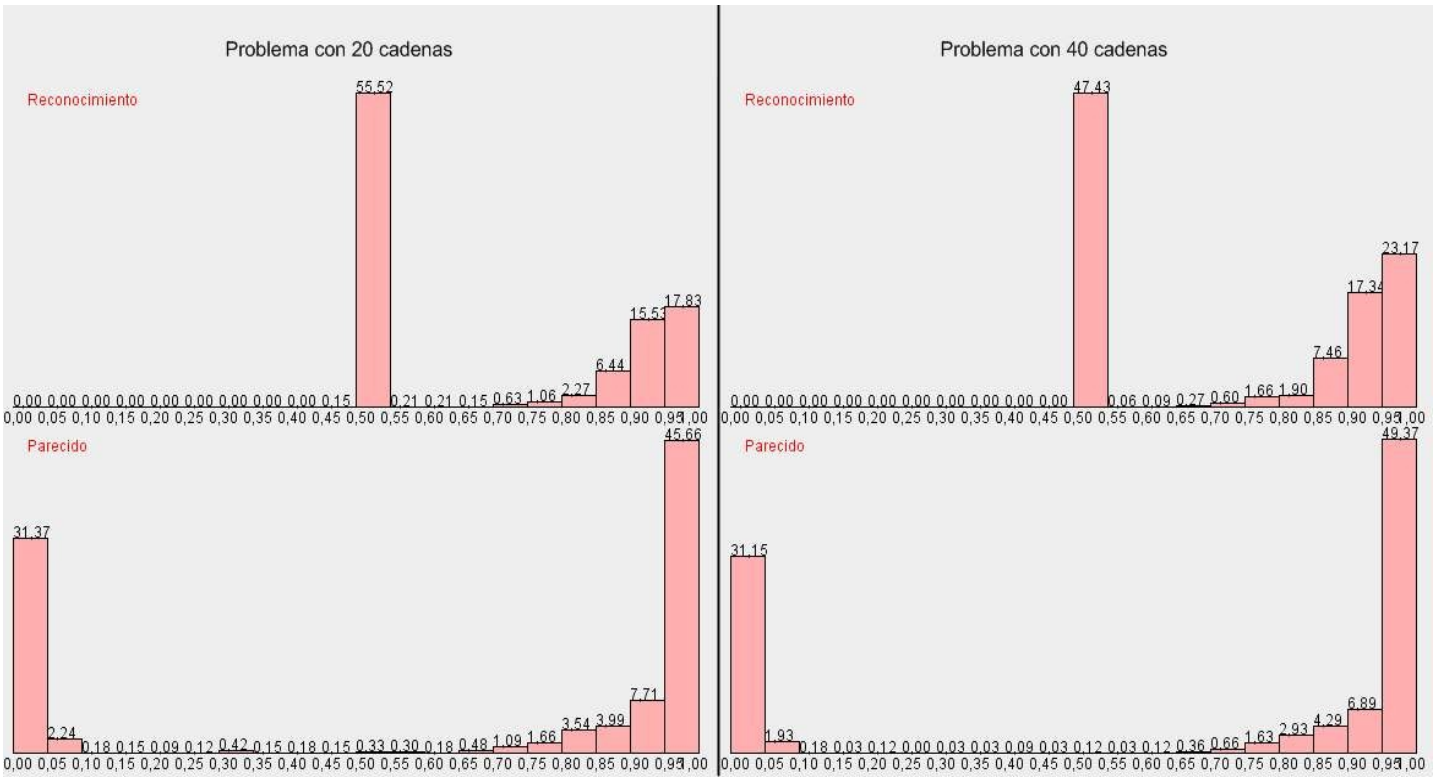
También en este caso podemos ver la distribución de los resultados para la mejor configuración:



Donde podemos ver claramente en la distribución la tendencia a obtener buenos resultados, con reconocimientos superiores al 90% en el 83% de los casos.

5.1.4 Estudio de la influencia del número de cadenas (problema B3)

En la base de datos añadimos el mismo problema con distinto número de cadenas cada uno, para poder determinar si esto tenía alguna consecuencia.



Como se puede ver en el gráfico, con un mayor número de cadenas, los resultados del algoritmo mejoran en general. En nuestras pruebas, las respuestas con un reconocimiento del lenguaje entre el 95% y 100% pasaron de ser un 17,83% a un 23,17% del total y además, las pruebas con un reconocimiento de entre el 50% y 55% (las que fallaron, dado que son las que aceptan o rechazan todas las cadenas) pasaron de un 55,52% a un 47%.

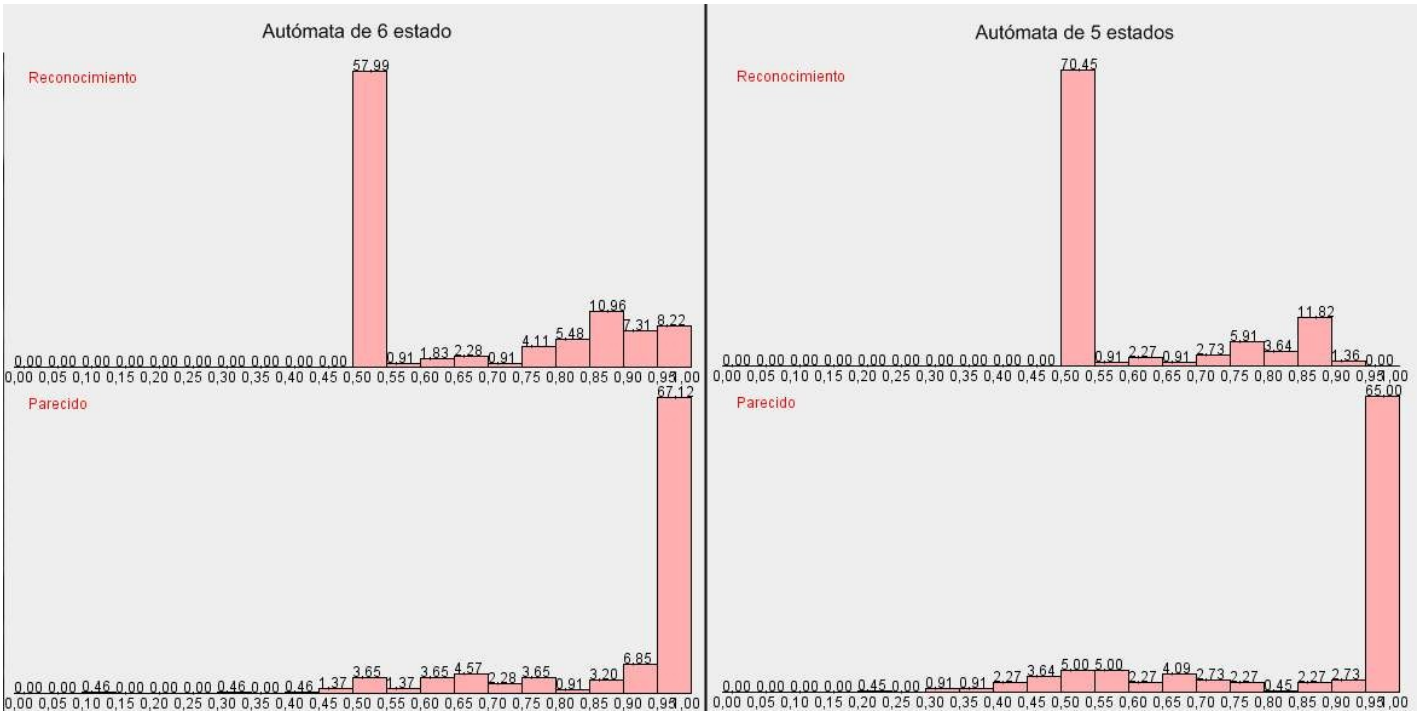
Ademas, si consideramos una de las configuraciones para las que se comporta bien el problema (por ejemplo, aquellas con el cruzador 1 y el mutador 3), los resultados son aún mejores, pasando los mejores resultados de ser el 21,27% al 34,98% mientras que los malos bajaron del 40,72% al 26,91%.

Esto nos lleva a pensar que un mayor número de cadenas puede ser bueno para nuestro algoritmo, dado que ayuda a guiar la búsqueda del AF. Combinando esto con la elección de la configuración correcta, la precisión y calidad del algoritmo es sobresaliente.

5.1.5 Estudio de las “configuraciones especiales” (problema B9)

Para estas pruebas, estudiamos lo que sucedería al tratar de encontrar una AFP de 5 o 6 estados para un conjunto de cadenas que sabíamos a priori que no podía ser descrito correctamente sino con un AF de 7 estados.

En este caso, sólo analizamos las configuraciones con el cruzador 1 y el mutador 3, y tenemos lo siguiente:



Vemos que el número de soluciones que dan malos resultados aumenta de 57,99% a 70,45%, lo que era de esperar. Sin embargo, incluso en el caso de que solo usamos 5 estados, hay un 11,82% de soluciones que reconocen entre el 85% y 90%.

Esto confirma las expectativas de que el algoritmo es razonablemente bueno para casos en los que el número de estados de la solución solicitada no es suficiente para describir completamente el lenguaje. Creemos que ésta es una de las características más interesantes de nuestro algoritmo.

5.2 Análisis de resultados

Consideramos que los resultados son en general muy representativos de un buen comportamiento del algoritmo en todas las condiciones estudiadas. Lamentablemente, no parece haber una configuración perfecta como esperábamos, pero creemos que la capacidad que le dimos a nuestra implementación para configurar todos los parámetros soluciona en la mayor parte este inconveniente.

Si nos centramos exclusivamente en los resultados de aquellas configuraciones que son especialmente buenas en la mayoría de los casos, podemos ver que los resultados que se ofrecen son muy satisfactorios y la probabilidad de encontrar un AF que reconozca el lenguaje dado en un 100% es muy alta.

Creemos que los resultados muestran la capacidad de los algoritmos genéticos para afrontar este problema mediante el uso de AFP, y estamos especialmente satisfechos con su capacidad para encontrar un buen AFP incluso cuando sabemos que encontrar uno perfecto es imposible.

5.3 Comparación con otros algoritmos

A continuación, explicaremos distintos algoritmos ya conocidos para resolver el mismo problema

que estamos tratando nosotros u otros problemas muy similares, adaptándolos a las caracterísitcas particulares de nuestro problema para poder realizar una comparación entre los resultados obtenidos de una y otra forma.

Los resultados no solo varían en cuanto al tiempo necesario para alcanzarlos, sino que las limitaciones de nuestro algoritmo y la de los demás son distintas y trataremos de resaltarlas.

5.3.1 Fuerza bruta

Para esta comparación, no tiene sentido realizar una implementación de un algoritmo de fuerza bruta, dado que esto sería prácticamente imposible ya que se trata de un problema NP-completo. Sin embargo, creemos que es interesante analizar el coste aproximado de un algoritmo de este tipo para reafirmar la utilidad del algoritmo desarrollado, ya que cualquier método óptimo para resolver el problema es, en el fondo, un refinamiento de la pura "fuerza bruta". Para ello, estudiaremos el número de autómatas que habría que analizar en el peor caso para dar con un autómata que cumpla las características del que estamos buscando.

Cálculo

En nuestro alfabeto, {0,1} tenemos 2 símbolos. Consideraremos que el estado inicial no lo tendremos en cuenta, y fijamos que el estado inicial sea siempre Q0. No es lógico que todos los estados de un autómata sean finales, y tampoco que ninguno sea final. En el primer caso, porque todas las cadenas serían reconocidas. En el segundo caso porque ninguna cadena sería reconocida. Por lo tanto, para las posibilidades de los estados que pueden ser finales, quitaremos estas dos posibilidades.

Para cada símbolo (0 o 1), tenemos n posibles destinos (los n-1 estados y el mismo origen). Por tanto, tenemos:

n^2 corresponde a las transiciones de 0 y 1 de cada estado. Como tenemos n estados, multiplicamos n^2, n veces, quedando n^{2n}. Pueden darse 2^n-2 posibilidades correspondientes a los estados finales (2^n-2, las 2 que consideramos anteriormente, que todos sean finales y que todos sean no finales).

Resultando:

$n^{2n} \cdot (2^n-2)$ autómatas finitos deterministas posibles de n estados.

Casos a considerar

Sabemos que en este cálculo existen muchos autómatas que son simétricos (iguales entre sí pero cambiando tan sólo el nombre de algún estado), y también hay autómatas inconexos (puede haber grupos de estados que se conecten con transiciones entre sí, pero que no haya forma de llegar a ellos).

Sin embargo, un algoritmo de fuerza bruta tendría que buscar entre estos autómatas. Si no fuera así, se perdería mucho tiempo en hacer comprobaciones del tipo: comprobar si un autómata es simétrico a alguno de los anteriores, o comprobar para cada autómata si hay estados inconexos, convirtiendo el algoritmo en algo muy costoso.

Por tanto, si queremos calcular cuántos autómatas habría que calcular con un algoritmo de fuerza bruta, es justo considerar todos los autómatas, aunque haya autómatas simétricos o inconexos entre ellos.

Conclusión

Veamos algunos cálculos, siendo n el número de estados:

n=5: 292.968.750 autómatas.

n=10: más de 10^{22} autómatas.

n=15: aproximadamente 10^{38} autómatas

n=20: aproximadamente 10^{58} autómatas.

Podemos observar que el número de autómatas es gigantesco, y que resulta inviable comprobar todos los autómatas finitos desde antes de llegar a 10 estados. Sin embargo, nuestro algoritmo, como se pudo comprobar en las pruebas, da buenos resultados hasta 12 estados trabajando con poblaciones de 4000 autómatas y 50 pasos como máximo, lo que quiere decir que solo se analizan 200000 o 2.10^6 autómatas y aunque los cálculos son más complejos el conste será muchísimo menor (y más teniendo en cuenta que el límite de 50 pasos no siempre se alcanza y que no siempre es necesario trabajar con una población de 4000 cuando una de 500 da buenos resultados)

5.3.2 Algoritmos "óptimos"

Para realizar estas pruebas, nos hemos basado en diferentes artículos que analizan problemas similares a los que tratamos en este proyecto. Para poder hacer las pruebas, decidimos implementar dos algoritmos distintos, adaptados para que funcionaran con las mismas entradas que el algoritmo genético.

Llamamos algoritmos "óptimos" a estos algoritmos porque dadas sus características deterministas siempre encuentran una solución (si existe y cumple ciertas condiciones) y esta solución es la mismas para los mismos datos de entrada. Esto no quiere decir que siempre encuentren el menor autómata posible para un problema dado.

Otra característica que tienen estos algoritmos es que, a pesar de ser iterativos, en cada iteración se consigue un AF válido para las cadenas de entrada.

Existen otros algoritmos para resolver este problema o similares que se podrían ajustar, pero consideramos que las dos implementaciones que se presentan a continuación son representativas del conjunto de algoritmos deterministas para llegar a un autómata mínimo o cuasi mínimo.

Algoritmo de búsqueda intensiva (voraz)

Este algoritmo fue diseñado por nosotros, aunque utilizando conceptos recogidos de diferentes artículos. Este algoritmo cumple las características que se dieron antes pero además encuentra siempre el autómata mínimo (si tiene tiempo suficiente).

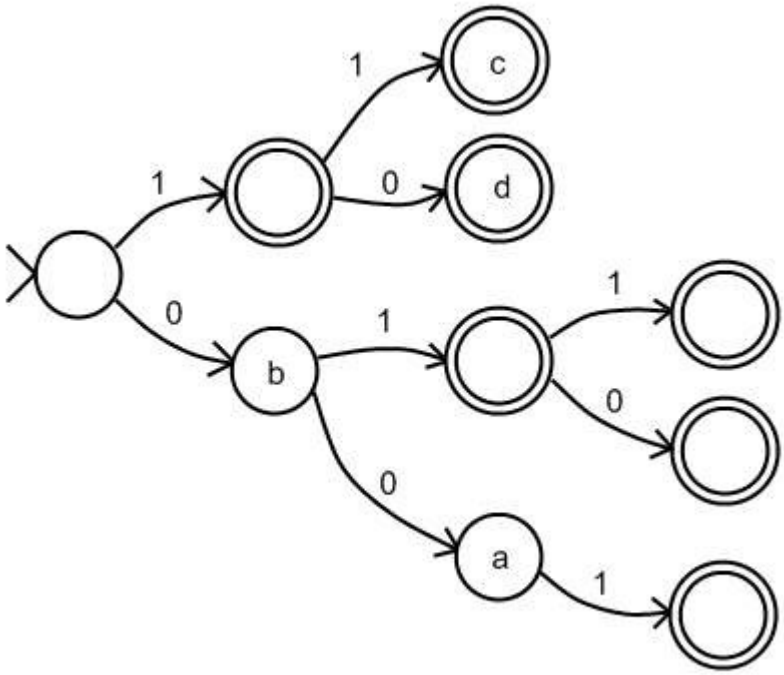
El mayor problema de este algoritmo es que su coste crece de forma exponencial con el número de cadenas de entrada y el largo de éstas. A continuación veremos su funcionamiento, donde esto quedará patente.

El algoritmo comienza generando un AF válido para reconocer el lenguaje a partir de las cadenas de entrada. Esto es simple, y funciona procesando cada una de las cadenas aceptadas (las rechazadas no son necesarias en este paso) y creando una rama que lleva a un estado de aceptación. Para que el autómata generado sea un AFD y no un AFND, las partes comunes de las ramas se unen automáticamente.

Por ejemplo, si tenemos las siguientes cadenas de entrada:

Aceptadas	**Rechazadas**
1	“”
01	0
10	00
11	000
011	0000
010	00000
001	

En el primer paso se creará el autómata:



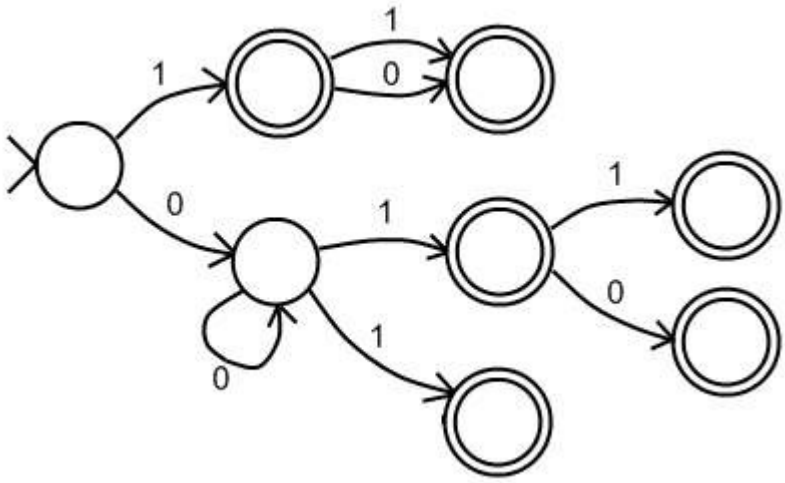
Como se puede ver, este autómata ya cumple las especificaciones, pero sin embargo tiene 11 estados frente a los 2 estrictamente necesarios para reconocer correctamente las cadenas del lenguaje dadas como ejemplo.

Para solucionar esto el algoritmo irá probando a unificar los estados por pares. Además, como una unificación puede afectar a otra, se utiliza un algoritmo de búsqueda para probar todas las posibles combinaciones. Tras cada unificación se comprueba que el autómata resultante sea válido (esto es, reconozca correctamente las cadenas dadas como ejemplo).

La unificación del estado 1 en el estado 0, consiste en cambiar todas las transiciones que iban al estado 1 por transiciones hacia el estado 0. Paralelamente, si en el estado 1 existe alguna transición para un valor de entrada no definido para el estado 0, esta transición se añade al estado 0. Como esto puede provocar que algunos estados queden inconexos del resto del autómata, al realizar la unificación también se comprueba esto y se procede a eliminar los estados que ya no sean

necesarios.

A continuación se muestra como quedaría el autómata del ejemplo anterior tras unificar el estado a con el b y el estado c con el d:



Como podemos ver, este autómata también es válido y tiene 2 estados menos.

Mediante este proceso se obtiene un autómata mínimo, aunque puede ser muy costoso cuando el autómata original tiene muchos estados (lo que depende de las cadenas aceptadas y rechazadas por el autómata.

Adaptación del algoritmo k-tails

El segundo algoritmo que implementamos es una adaptación del algoritmo conocido como k-tails a nuestro problema. Decimos adaptación porque el algoritmo esta definido teóricamente para un problema similar pero no igual al que planteamos. Sin embargo, nuestra implementación tiene las mismas ventajas y problemas que el algoritmo original: que es menos costoso que el algoritmo planteado anteriormente, pero que no siempre encuentra el autómata mínimo.

Este algoritmo es muy similar al anterior, pero en lugar de unificar todos los estados entre sí, utiliza una "función de discriminación" o las k-tails (cadenas reconocidas de k caracteres desde un estado) para determinar la compatibilidad entre estados. El problema de este algoritmo es que hay veces que determina que dos estados son incompatibles cuando realmente no lo son (en general, por lo que pudimos experimentar, debido a que no es capaz de predecir las consecuencias de los bucles que pueden resultar de una unificación).

En nuestro caso, como conocemos las cadenas aceptadas y rechazadas y sólo el tratamiento que haga el autómata de éstas es lo que nos interesa, sólo utilizamos éstas para generar las tails o colas de los estados. En particular, utilizamos todas las posibles sub-cadenas que se puedan formar a partir de las cadenas aceptadas y rechazadas.

En cada estado sabremos una serie de cosas:

- Las subcadenas que se deben aceptar desde un estado i para que el autómata sea válido (AV_i)
- Las subcadenas que se deben rechazar desde un estado i para que este sea válido (RV_i)
- Todas las subcadenas que se aceptan desde un estado i (AT_i)
- Todas las subcadenas que se rechazan desde un estado i (RT_i)

Entonces el estado 0 se puede unificar con el estado 1 si y solo si:

$$\forall c \subset AV_0 \Rightarrow c \in AT_1 \vee c \notin RV_1$$
$$RV_0 \cap AT_1 = \emptyset$$

También se pueden añadir otras restricciones, permitiendo que un estado no final se unifique con uno final considerando la cadena "" como un caso especial.

Comparación

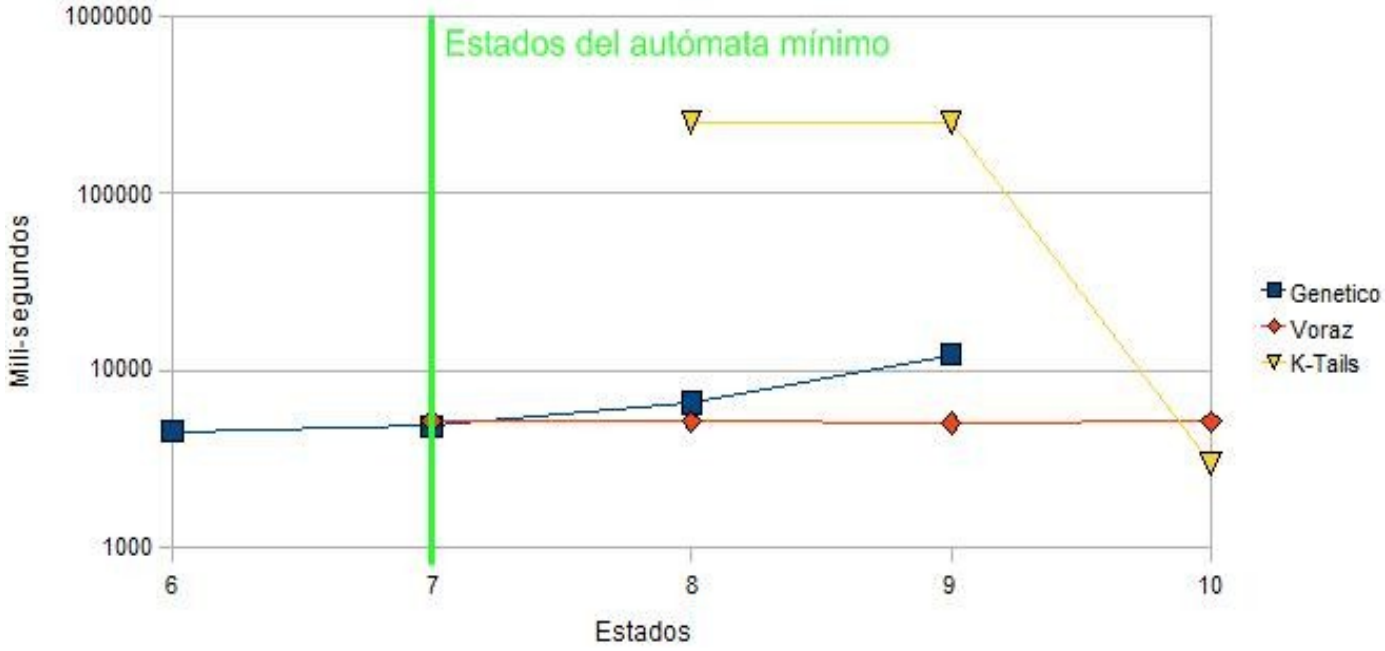
Aunque los algoritmos se han adaptado para trabajar con el mismo problema, el enfoque es muy diferente lo que hace muy difícil compararlos.

La mayor diferencia es que los algoritmos estudiados funcionan creando un autómata y luego reduciéndolo. Esto implica que a mayor número de estados que le pidamos al algoritmo, menos va a tardar, a diferencia de nuestro algoritmo que se beneficia un menor número de estados elegido porque esto reduce el tiempo de ejecución.

Además, como vimos en la pruebas anteriores, nuestro algoritmo se beneficia de un mayor número de cadenas de ejemplos y da mejores resultados. Estos algoritmos, al tener un mayor número de cadenas de entrada tienen un mayor número de estados desde el comienzo y por lo tanto tardan más en encontrar una solución de N estados.

Todas estas diferencias hacen que nuestro algoritmo, incluso si tardara más en tiempo, pueda tener aplicaciones en situaciones donde las características descritas sean beneficiosas. En particular, no encontramos ningún otro algoritmo que permita utilizar un N menor que el número mínimo y dar el mejor resultado para éste y nuestro algoritmo es claramente capaz de esto.

Sin embargo, a continuación mostramos los tiempos que requieren los distintos algoritmos aplicados al mismo problema y trataremos de llegar a una conclusión general.



Como podemos ver en este ejemplo, los tiempos varían mucho con el número de estados que ponemos en cada uno de los algoritmos. Cada uno de ellos se comporta mejor en ciertas condiciones.

Con un número de estados bastante mayor que el necesario, el algoritmo k-tails resulta el mejor en tiempo. Sin embargo, según va bajando el número de estados, éste empeora bastante. De hecho, en este ejemplo, no consigue encontrar el autómata mínimo cuando se exige que el número de estados sea 7. El algoritmo voraz se comporta muy bien para varios números de estados. Nuestro algoritmo genético, tarda un tiempo casi idéntico al voraz en encontrar el autómata mínimo. Y es el algoritmo genético el único capaz de encontrar un autómata razonablemente bueno con un número de estados inferior al requerido por el autómata mínimo.

En este otro caso, tenemos otra vez diferentes tiempos y distintas situaciones en las que cada algoritmo se comporta mejor.

En este ejemplo, el algoritmo genético es el que mejor se comporta para buscar el autómata mínimo. Cuando se ofrecen más estados de los necesarios para el autómata mínimo, mejoran los tiempos de los algoritmos óptimos.

Con esto podemos decir que nuestro algoritmo se comporta mejor en muchas circunstancias, lo que lo hace competitivo frente a los otros algoritmos que existen para solucionar el mismo problema.

6 Conclusiones

Como comentamos al comienzo de este proyecto, nuestros objetivos eran:

- Analizar, estudiar, desarrollar y probar un algoritmo genético para la generación de autómatas a partir de ejemplos (aprendizaje pasivo)
- Crear una aplicación que permita aplicar el algoritmo, estudiar distintas situaciones y principalmente permitir hacer pruebas masivas en un entorno distribuido

Creemos que cumplimos en la mayor medida ambos objetivos.

En lo que respecta al segundo, la aplicación funciona correctamente y nos permitió realizar de forma distribuida, ordenada y muy útil para el análisis, **más de 70.000 pruebas con el algoritmo genético** sobre una variedad de problemas y con distintas configuraciones. Consideramos que este número de pruebas es suficientemente grande para cumplir el primer objetivo, pero el sistema es capaz de hacer muchas más pruebas.

El haber creado el sistema distribuido nos permitió realizar las pruebas incluso más rápido de lo que nos esperábamos, teniendo varios ordenadores funcionando al mismo tiempo distribuidos por Internet. **El crear el sistema distribuido no fue una tarea sin problemas**, la mayoría de índole tecnológico, que pudimos solucionar, y algunos incluso evitar, gracias a la elección de tecnologías adecuadas y muy desarrolladas. Para conseguir esto, creemos que la experiencia previa fue nuestra mejor herramienta, pero el realizar prototipos, comparar distintas opciones y, en general, obtener la mayor cantidad de información posible antes de comenzar el desarrollo fueron elementos clave del éxito del proyecto.

Sobre el primer objetivo, podemos decir que estamos muy conformes con los resultados alcanzados. El análisis del algoritmo fue amplio, permitiéndonos crear los distintos mutadores y cruzadores y determinar la relevancia desde un punto de vista científico del trabajo desarrollado. **El estudio fue profundo, permitiéndonos realizar una implementación que tuviera en cuenta las características particulares** y nos permitiera realizar todas las pruebas que consideráramos necesarias. El desarrollo e implementación del algoritmo fueron muy satisfactorios, permitiéndonos aplicar distintas optimizaciones y corregir errores hasta conseguir una aplicación muy completa y funcional. Por último, **las pruebas sobre el algoritmo fueron extensas y dieron unos resultados muy satisfactorios.**

En el camino hacia alcanzar el primer objetivo, nos encontramos con diferentes inconvenientes. Desde el gran número de formas de implementar mutadores, cruzadores y calculadores de bondad, y la consecuente cantidad de pruebas necesarias para poder determinar cuál era la combinación conveniente, hasta conseguir la corrección en la implementación del algoritmo en sí (numerosas pruebas, revisiones e incluso reescritura de las distintas partes). También fueron problemáticas las decisiones de diseño, como por ejemplo el utilizar AFPs, que debían tomarse a priori y mantenerse a lo largo del proyecto. Afortunadamente, las decisiones tomadas parecen haber dado buenos resultados.

En lo que atiene al segundo objetivo, a los resultados, podemos ver que **el algoritmo cumplió nuestras expectativas**. Como era de esperar tiene ventajas y desventajas sobre otros algoritmos, pero creemos que hemos analizado cada una de ellas de forma que nos podemos hacer una idea de la utilidad del algoritmo y sus capacidades para ser aplicado y el interés de seguir estudiando su aplicación en otras partes de la Teoría de Autómatas.

Con todo esto, lo que queremos decir es que los objetivos que nos planteamos al comienzo de este proyecto fueron alcanzados con creces y estamos muy satisfechos con el trabajo realizado.

7 Posibilidades para futuros desarrollos

Durante el desarrollo del proyecto, surgieron muchas ideas que podrían aprovechar parte de la infraestructura desarrollada o que simplemente continuaban la línea de investigación adoptada.

Por ejemplo, el algoritmo genético se podría extender para aplicarlo sobre Autómatas de pila (APs) y dado que estos tienen una mayor capacidad descriptiva, sería interesante comparar los resultados y ver si este tipo de algoritmo sigue siendo útil para este tipo de autómatas. Lamentablemente, por restricciones de tiempo, no pudimos realizar una implementación de esto, pero el diseño del servidor está preparado para añadir este tipo de autómatas de forma muy sencilla.

Si los resultados fueran satisfactorios con APs, también se podría intentar hacer una implemantación de un algoritmo genético para aprender Máquinas de Turing (MTs) acercándonos por otro camino al objetivo inicial del proyecto y, posiblemente, obteniendo información muy interesante desde el punto de vista de la Teoría de Autómatas. Sin embargo, por las características de los autómatas (en particular, la dificultad para describirlos) y por la capacidad descriptiva de éstos, cabe esperar que a medida que subimos en la jerarquía de Chomsky los resultados empeoren. En todo caso, obtener dicho resultado (el empeoramiento a medida que subimos en la jerarquía) podría tener interés teórico en sí mismo y por ahora es sólo nuestra conjetura.

El desarrollo realizado también es interesante porque la misma filosofía, y parte del código, utilizado para la experimentación masiva y distribuida, podría ser usado para otro tipo de experimentos. Existen muchos algoritmos estocásticos como el nuestro, y esta forma de realizar pruebas puede aportar valiosa información estadística sobre su funcionamiento.

También es interesante la posibilidad de utilizar este enfoque para resolver problemas más prácticos, en los que normalmente se aplican otros algoritmos de aprendizaje sobre otras estructuras (como redes neuronales), como pueden ser el reconocimiento de caracteres, dado que creemos que este trabajo muestra la gran capacidad de los algoritmos genéticos para el aprendizaje de autómatas que pueden ser utilizados con estos fines.

Creemos que todas estas posibilidades aportan un valor extra a nuestro proyecto a pesar de que no pudiéramos incluirlas en él.

8 Bibiliografía

[1] J.E. Hopcroft, R. Motwani, J.D. Ullman: *Introducción a la teoría de autómatas, lenguajes y computación*, Pearson Educacion 2004.

[2] D. Kelley: *Teoría de autómatas y lenguajes formales*, Prentice Hall, 2001

[3] K.P. Murphy: *Passive Learning Finite Automata* (November 1995).

[4] J.L. Balcázar, J. Díaz, R. Gavaldà, O. Watanabe: *Algorithms for Learning Finite Automata from Queries: A Unified View*, (September 1996).

[5] I. Rodríguez, M. Merayo, M. Núñez, HOTL: Hypotheses and observations testing logic, *Journal of Logic and Algebraic Programming,* 74, 57-93, Elsevier.

[6] L. Miclet: *Regular Inference with a Tail-Clustering Method* (November 1980).

[7] K.J. Lang: *Random DFAs can be Approximately Learned from Sparse Uniform Examples* (October 1998).

[8] B. Sotomayor, L. Childers: *Globus Toolkit 4: programming Java services*

[9] V. Silva: *Grid Computing for Developers*

[10] F. Berman, G. Fox, A. Hey: *Grid Computing Making the Global Intrastructure a Reality*

[11] Sitio oficial de BOINC.(http://boinc.berkeley.edu/)

[12] Comunidad de JBoss.(www.jboss.org/)

[13] J.J. Merelo Guervós: *Informática Evolutiva: Algoritmos genéticos* (http://geneura.ugr.es/~jmerelo/ie/ags.htm). Universidad de Granada.

[14] W. Tzeng: *Learning Probabilistic Automata and Markov Chains via Queries.* (1992)

Apéndice A: Manual de uso del cliente

1. Funcionalidad

Esta aplicación permite resolver problemas de la base de datos del servidor. Los resultados son enviados de nuevo al servidor.

2. Requisitos mínimos

Se debe disponer de un sistema operativo con Java 1.6 o posterior. Probado utilizando Windows 2000, XP y Vista.

3. Instrucciones de instalación

El contendido del fichero cliente.zip debe extraerse en una carpeta y ejecutar el fichero Cliente.bat (en sistemas operativos distintos de Windows se debe crear un fichero batch que ejecute los comandos equivalentes).

Debido a problemas con Windows, puede ser necesario que la dirección de la carpeta no tenga espacios (no se recomienda instalar en “Mis documentos” o en “Archivos de programas” o similares).

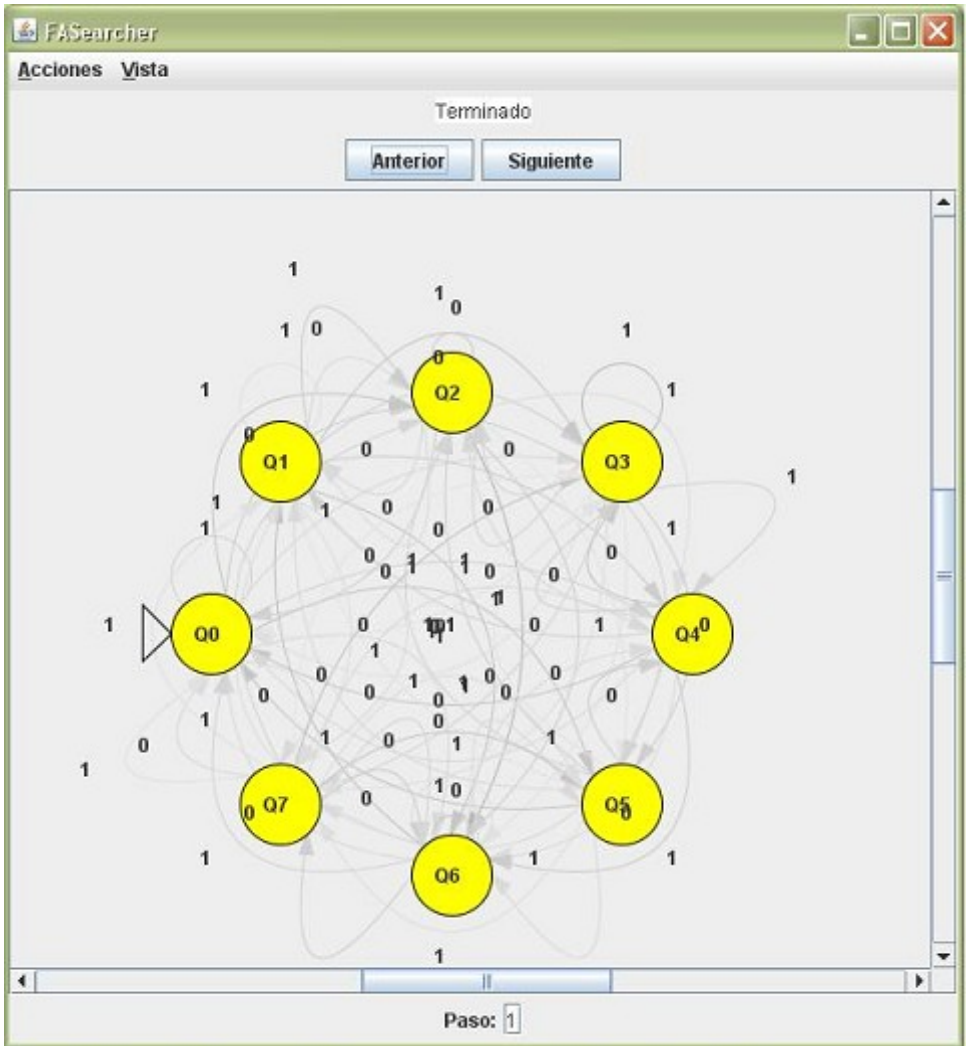
4. Instrucciones de uso

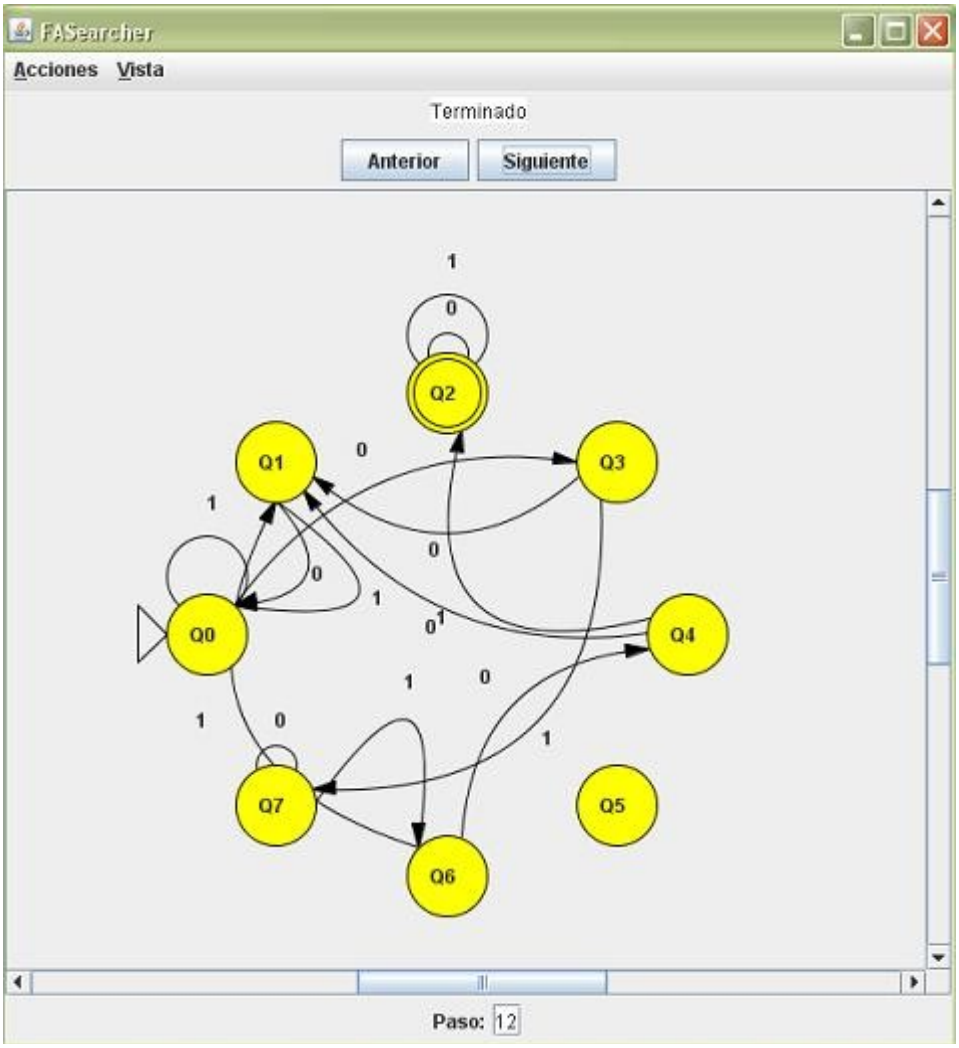
Al ejecutar la aplicación aparecerá una ventana. Esta ventana dispone de dos menús.

4.1. Menú Acciones

4.1.1. Resolver un problema

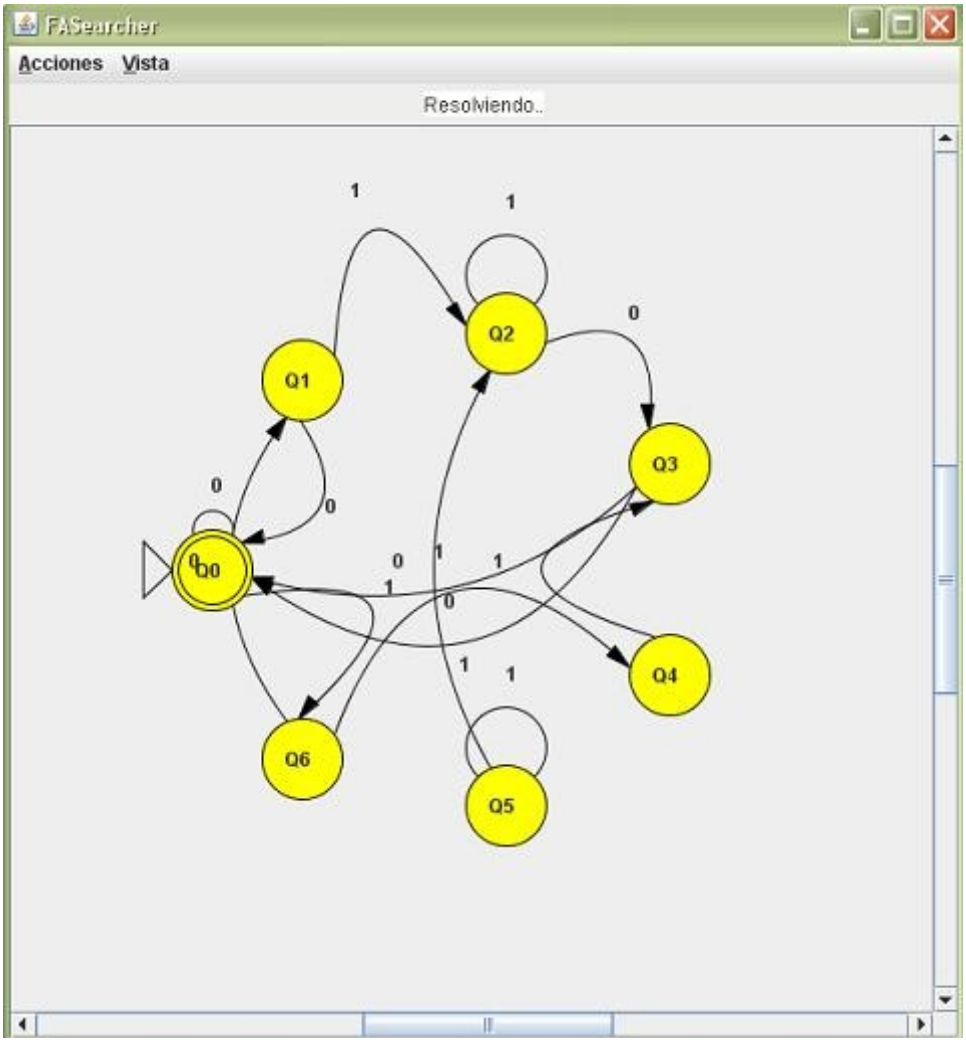
Si hace clic en Resolver un problema, la aplicación cogerá un problema de la base de datos del servidor y lo resolverá. Después, el cliente verá la solución. La solución estará formada por un conjunto de autómatas finitos probabilistas (AFPs). Verá el mejor de cada iteración conseguido. Para ir viendo cada autómata, haga clic en siguiente, para ver el autómata del siguiente paso. Cuando el número de paso no avance, habrá llegado al último autómata conseguido.


FASearcher
Acciones Vista
Terminado
Anterior
Siguiente
Q0
Q1
Q2
Q3
Q4
Q5
Q6
Q7
Paso: 1



4.1.2. Empezar a resolver problemas

Si hace clic en Empezar a resolver problemas, la aplicación empezará de forma automática a resolver problemas de la base de datos. Cuando termine de resolver un problema, enviará los resultados a la base de datos del servidor. Esto será automático y transparente al usuario. Cuando se termina de resolver un problema, se empieza automáticamente con el siguiente. El usuario irá viendo en la ventana los mejores autómatas que se van consiguiendo y se envían de vuelta a la base de datos.



4.1.3. Parar de resolver problemas

Si hace clic en Parar de resolver problemas, se detendrá el proceso de resolver problemas. Se cancelarán los resultados del último problema en ejecución, y no se enviarán más problemas de la base de datos.

4.2. Menú Vista

El menú Vista contiene la opción Mostrar autómatas generados.

4.2.1. Mostrar autómatas generados

Si hace clic en Mostrar autómatas generados, se activará o se desactivará la opción de ver los autómatas que se van generando.

Apéndice B: Manual de uso del administrador

1. Funcionalidad

Esta aplicación es la que se encarga de gestionar los problemas de la base de datos del servidor. Puede crear nuevos problemas, editando sus configuraciones, borrarlos de la base de datos, ver las estadísticas de todos los problemas... Dispone de muchos más privilegios que la aplicación cliente.

2. Instrucciones de instalación

El contendido del fichero cliente.zip debe extraerse en una carpeta y ejecutar el fichero Cliente.bat (en sistemas operativos distintos de Windows se debe crear un fichero batch que ejecute los comandos equivalentes).

Debido a problemas con Windows, puede ser necesario que la dirección de la carpeta no tenga espacios (no se recomienda instalar en "Mis documentos" o en "Archivos de programas" o similares).

3. Requisitos

Se debe disponer de un sistema operativo con Java 1.6 o posterior. Probado utilizando Windows 2000, XP y Vista.

4. Instrucciones de uso

Aparece una ventana que muestras cuatro menús y una barra de herramientas a la izquierda. Los menús permiten ejecutar las acciones seleccionadas, y la barra de herramientas de dibujo permite dibujar un autómata.

4.1. Barra de herramientas de dibujo

Esta barra sólo servirá de utilidad cuando se esté dibujando un autómata. Para ello, habrá que hacer clic en Problemas->Dibujar Autómata.

Cada uno de los iconos de la barra permite insertar un elemento o editar el dibujo de la siguiente forma:

4.1.1. Seleccionar y mover

Su icono tiene la forma de una mano. Permite entrar en el modo de selección. En este modo, se pueden mover los estados de los autómatas libremente por el panel.

4.1.2. Crear nuevo estado

El icono es un círculo. Permite entrar en el modo de crear estados. En este modo, cada clic en el panel creará un estado nuevo. Los estados serán creados con el nombre Q0, Q1, así sucesivamente

4.1.3. Crear nueva transición

Viene representado por una flecha. Permite entrar en el modo de insertar transición. En este modo, se hace clic sobre un primer estado, del que parte la transición, y después se espera que se haga clic en un segundo estado, que es el destino de la transición. Cuando se haga el segundo clic, aparecerá un mensaje que pedirá al usuario que elija el valor 0 o el valor 1 para esa transición.

Si se quiere modificar el valor de una transición, basta hacer clic de nuevo en el estado origen del

que parte la transición, y hacer clic en el nuevo estado destino, eligiendo el valor correcto para la transición.

4.1.4. Activar estado final

El icono es un doble círculo. Permite entrar en el modo de selección de estado final. En este modo, al hacer clic sobre un estado, se activa como estado final o aceptación (representado con un doble círculo a su alrededor), o se desactiva si era estado final (queda como un círculo simple).

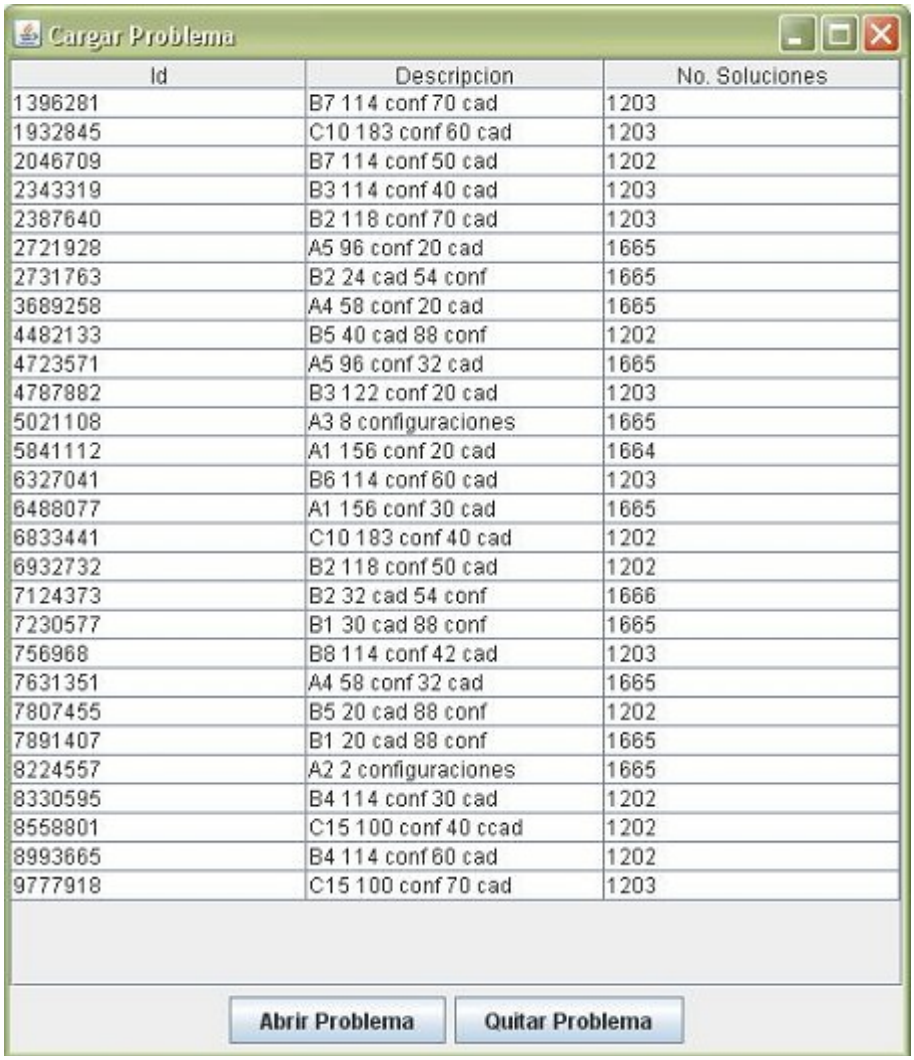
4.2. Menú Problema

Este menú contiene las opciones relacionadas con la gestión de problemas. Éstas incluyen: consultar problemas resueltos, crear nuevos problemas, consultar las estadísticas, ver las soluciones...

4.2.1. Gestionar problemas

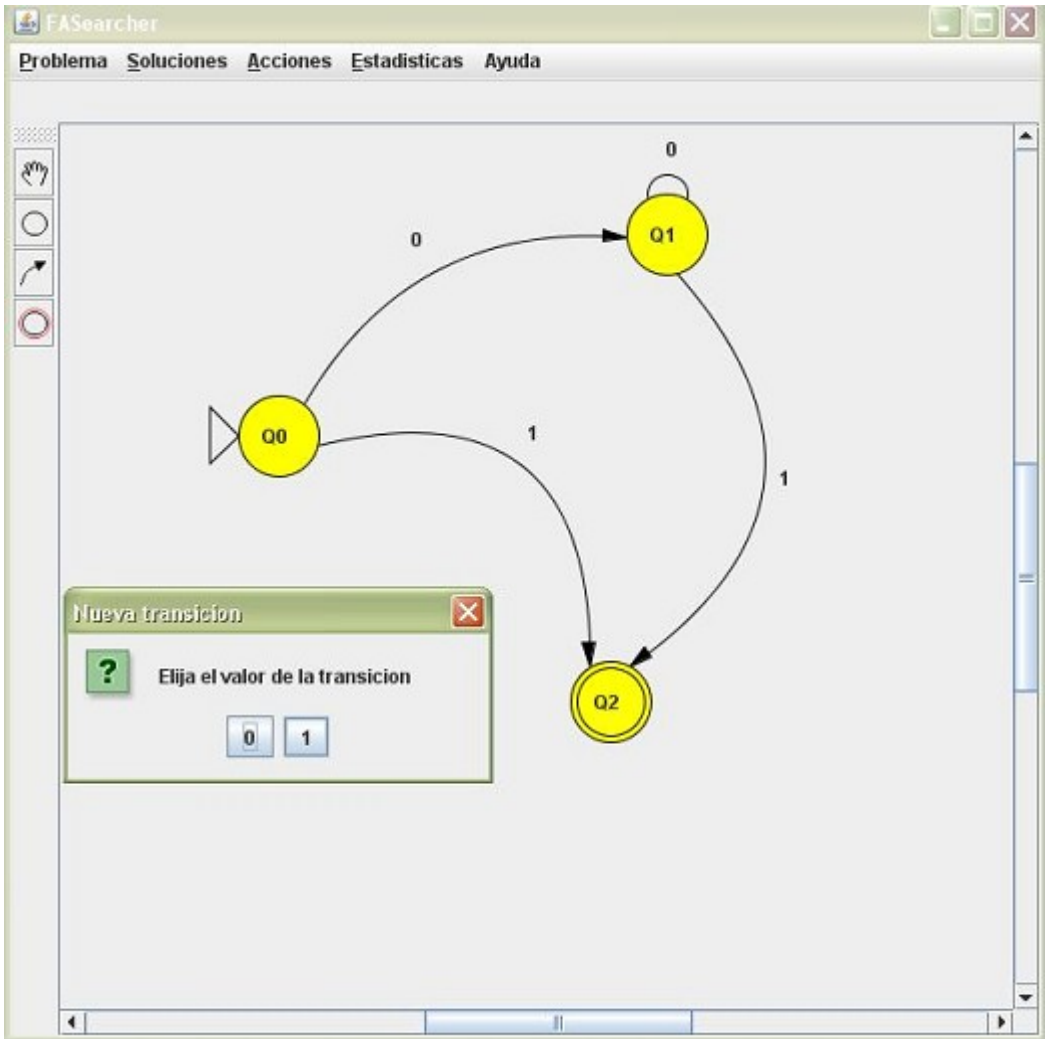
Esta opción abre una pantalla donde se muestran los problemas que contiene la base de datos. Estos problemas tienen: un identificador, una descripción del problema, y el número de soluciones de las que se dispone. Hay dos opciones posibles:

- Borrar problema. Elimina el problema de la base de datos definitivamente.
- Abrir problema. Abre el problema de la base de datos. Esto supone que se muestra el autómata de entrada, se cargan las cadenas que tenía de entrada.



4.2.2. Dibujar autómata

Esta opción permite dibujar sobre el panel. En este momento se puede utilizar la barra de herramientas de dibujo que aparece en la parte izquierda de la pantalla. Si ya aparecía un autómata dibujado, al hacer clic sobre Dibujar autómata se borrará el antiguo autómata y sus datos asociados (no aparecerán cadenas en las listas de aceptadas y rechazadas y no habrá configuraciones del anterior problema).



4.2.3. Generar autómata aleatorio

Esta opción permite dibujar un autómata aleatorio. Al hacer clic sobre la opción, aparecerá un mensaje que pedirá el número de estados que se desea que tenga el autómata aleatorio creado. El autómata generado será un Autómata Finito Determinista (por tanto todas las transiciones tendrán un estado destino). Este autómata tendrá por lo menos un estado final, y por lo no menos un estado no final.

4.2.4. Generar cadenas aleatorias

Esta opción permite crear una lista de cadenas aleatorias, tanto cadenas aceptadas como rechazadas, que serán añadidas a las listas que luego formarán parte de la entrada del problema cuando se vaya a resolver.

Si se ejecuta esta opción varias veces desde el mismo autómata, en vez de sustituir las listas de cadenas anteriores, cada vez aparecerá un cuadro de diálogo que preguntará si se desea añadir las nuevas cadenas creadas a las listas de cadenas creadas anteriormente.

4.2.5. Manipular cadenas

Esta opción abre una ventana en la que aparecen dos paneles. En el de la izquierda, aparecen las

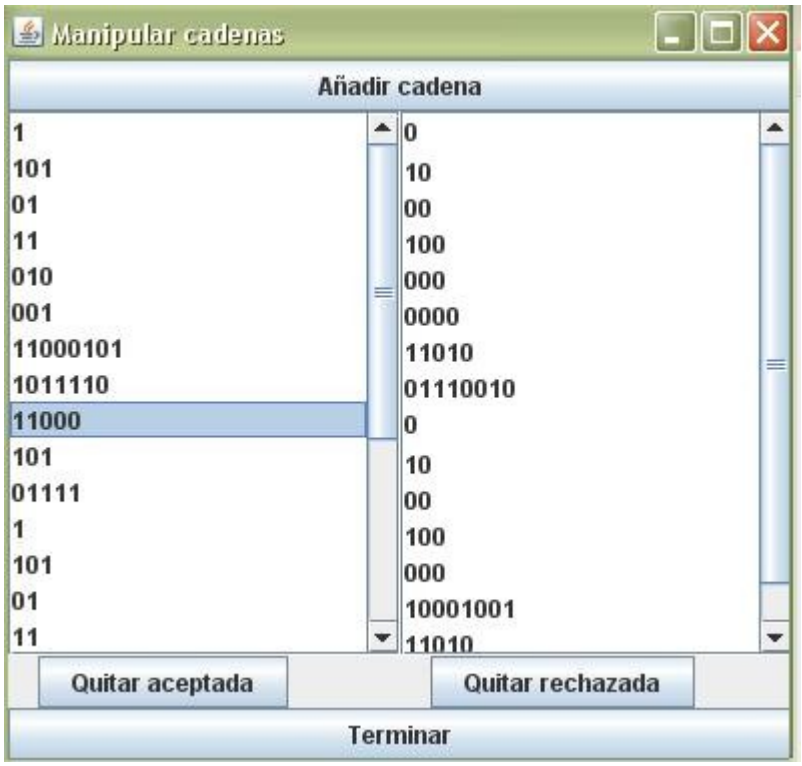
palabras que deberán ser aceptadas por el autómata cuando se pida resolver el problema. En el de la derecha, aparecen las palabras que deberán ser rechazadas por el autómata, al resolver el problema.

Se permite introducir varias veces la misma cadena de cualquiera de las dos listas. Esto supone que la entrada para el problema que se resolverá contendrá cadenas repetidas.

Las utilidades de esta función son: que se puede dar más importancia a una cadena que a otra, y que en problemas en los que alguna de las dos listas disponga de pocos ejemplos, el número de éstos puede ser aumentado simplemente volviendo a introducir la misma cadena. Sin embargo, esto sólo tiene sentido hacerlo sobre un número limitado de cadenas, ya que si introdujéramos varias veces todas las cadenas, este efecto desparecería.

Consejo al usuario:

Es importante beneficiarse de esta ventaja en problemas que tengan muy pocas palabras aceptadas o muy pocas rechazadas. En la lista que contenga muy pocos ejemplares, bastará con introducir varias veces la misma cadena para aumentar el rendimiento de la resolución del problema considerablemente, puesto que el algoritmo que busca el algoritmo será más correcto cuantas más cadenas tenga como ejemplos de palabras aceptadas y rechazadas.



4.2.6. Modificar configuraciones

Al ejecutar esta opción del menú, se abrirá un panel en el que se pueden modificar las distintas configuraciones de los problemas. Las opciones disponibles son: introducir una nueva configuración, o borrar una configuración existente.

Al crear una configuración, deberán introducirse los siguientes parámetros para el algoritmo genético:

Número de estados: el número de estados que tendrá el autómata finito probabilista que se mostrará como solución.

(valor típico: a juzgar por el usuario)

Mantener: el número de individuos que se deben mantener de una iteración del algoritmo a otra.

(valor típico: 20-60)

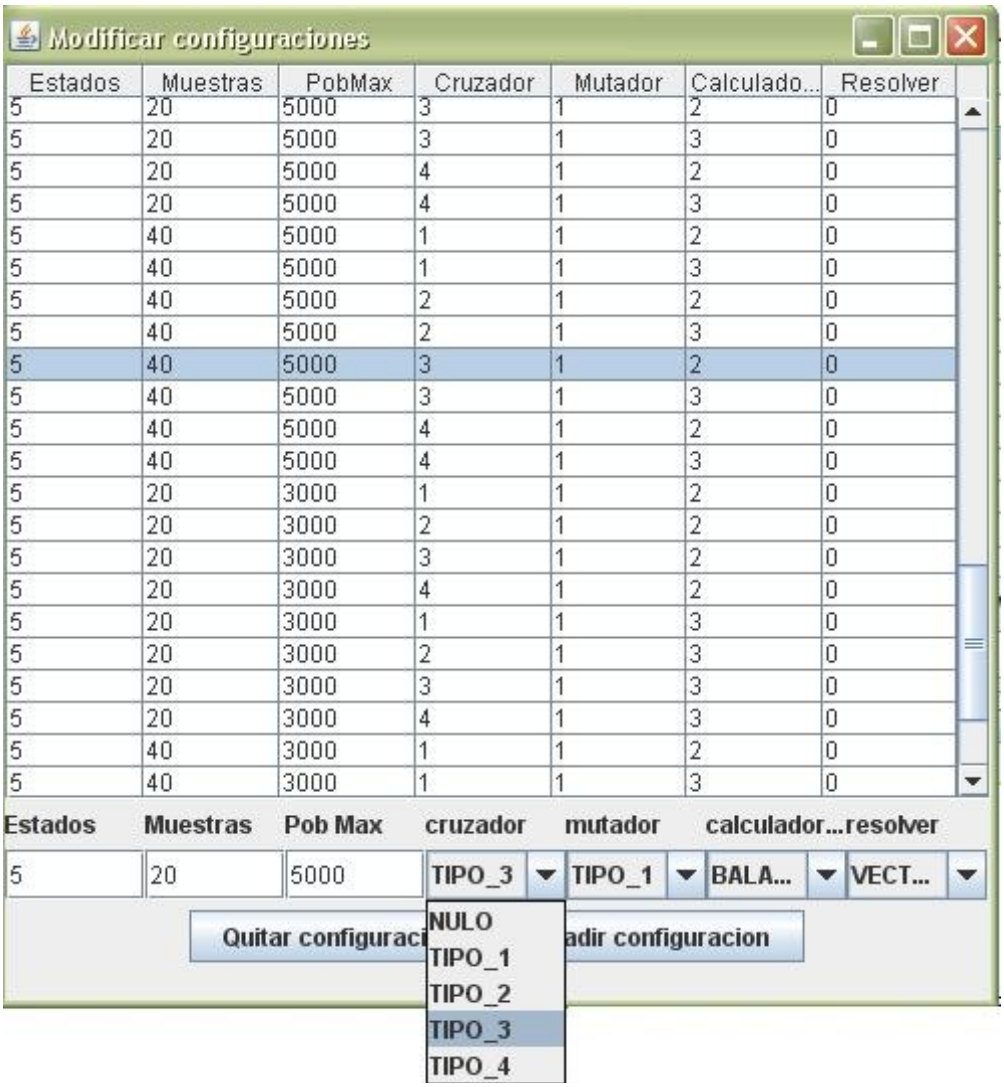
Población máxima: el número máximo de individuos que se generarán en cada iteración del algoritmo, dando lugar a la población de esa iteración.

(valor típico: 500-5000)

Cruzador: el cruzador que se utilizará en la ejecución del algoritmo. Se dispone de 4 cruzadores diferentes, y un cruzador nulo. Este último cruzador simplemente no cruzará individuos. El objetivo de que se pueda seleccionar este último es comparar cómo sería el resultado si no hubiera un cruzador implementado en el sistema.

Mutador: el mutador que se utilizará en la ejecución del algoritmo. Se dispone de 3 mutadores diferentes y un mutador nulo. Al igual que con el cruzador, el mutador nulo simplemente no muta al individuo. Y el objetivo de sus existencia es análogo.

Calculador de bondad: el calculador de bondad que se utilizará en la ejecución del algoritmo. La aplicación dispone de 4 calculadores de bondad diferentes.



4.2.7. Enviar problema

Esta acción consiste en enviar el problema al servidor. Se enviarán los datos siguientes:

- Los datos del autómata dibujado (número de estados, transiciones y estados finales).
- Configuraciones creadas en el menú Modificar Configuraciones.
- Cadenas de entrada creadas mediante Generar cadenas aleatorias y Manipular cadenas.

4.3. Menú Soluciones

En este menú aparecen las opciones relacionadas con las soluciones a los problemas.

4.3.1. Explorar soluciones

Esta opción mostrará un panel con todas las configuraciones del problema cargado. A continuación, haga clic sobre Mostrar soluciones para ver todas las soluciones de ese problema. Aparecerá una lista de todas las soluciones a ese problema. Si se selecciona una configuración concreta, se pueden mostrar solamente las soluciones obtenidas con esa configuración. Para ello, haga clic en Mostrar

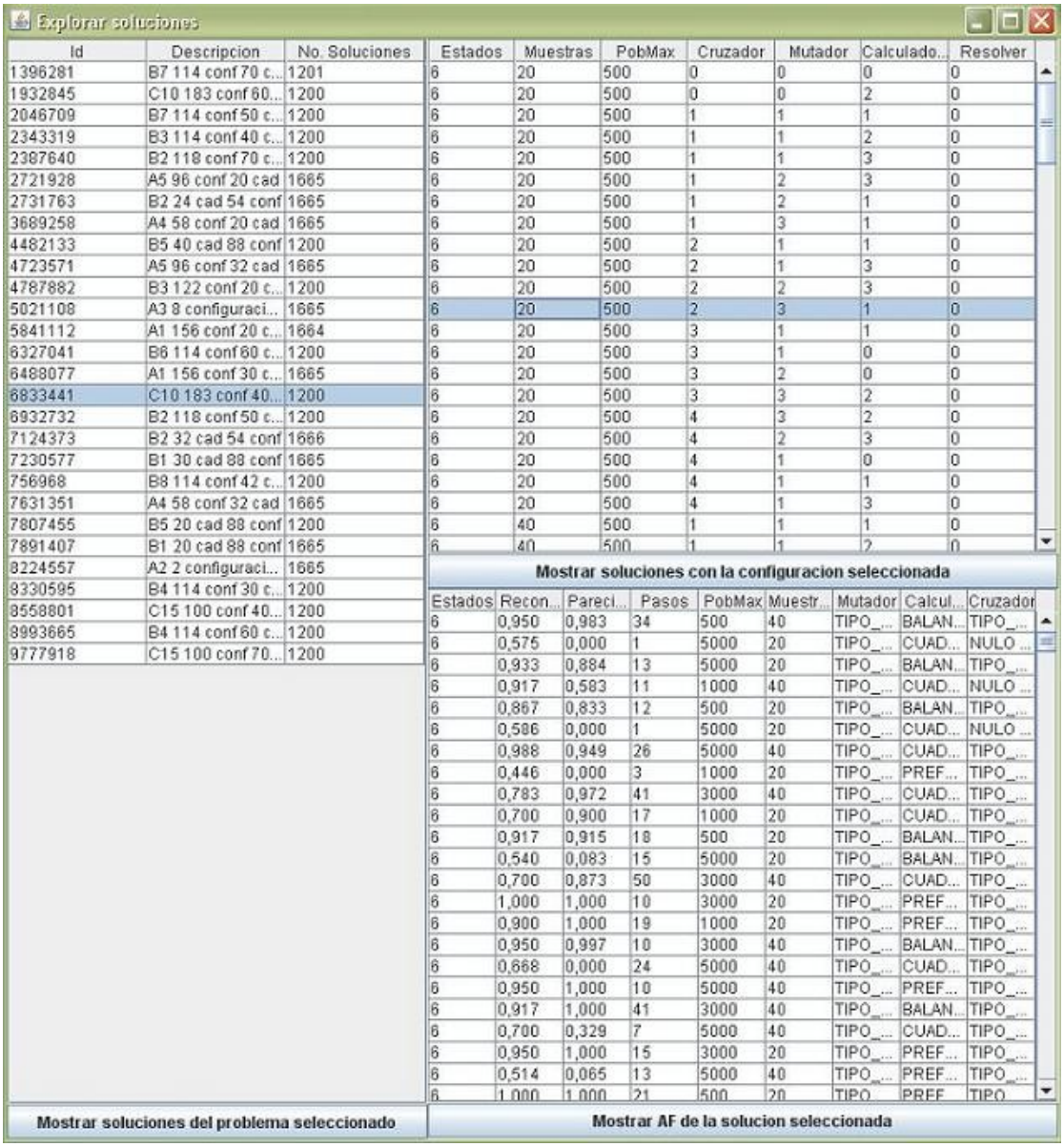
soluciones de configuración.

Los datos que muestra la solución son los siguientes:

Reconocimiento: es el porcentaje (de 0 a 1) de acierto al reconocer las cadenas aceptadas y rechazadas que ha obtenido esa solución. Si tiene un valor muy próximo a 1, el autómata cumple el objetivo deseado, ya que reconoce las cadenas que ha pedido el usuario.

Parecido AF: es el porcentaje (de 0 a 1) de acierto al ir transformando el Autómata Finito Probabilista en Autómata Finito Determinista.

Si se desea ver el Autómata Finito Probabilista solución obtenido, haga clic sobre Mostrar AF seleccionado.



4.4. Menú Acciones

Este menú muestra las opciones relacionadas con la resolución de los problemas.

4.4.1. Resolver problema actual

Esta opción resuelve el problema planteado. Los datos que toma son las cadenas creadas mediante las opciones Generar cadenas aleatorias, y Manipular cadenas. Al hacer clic sobre la opción, se deberá introducir la configuración con la que se desea resolver el problema, de forma similar a como se hace en la opción de Modificar configuraciones.

Cuando el programa toma estos datos, se dispondrá a resolver el problema.

Cuando termine, nos mostrará el Autómata Finito Probabilista resultado. Este se verá representado como un autómata dibujado mediante Dibujar autómata. Sin embargo, al ser un Autómata Finito Probabilista, las transiciones tienen probabilidades asignadas. Como estos valores no interesan exactamente al usuario, las transiciones se representadas con una escala de grises. Las probabilidades más cercanas al 1 se representarán con negro, como en el autómata finito determinista que se puede dibujar. Sin embargo, si la probabilidad se acerca al 0, el color de la transición se acercará más al blanco. De este modo, si un Autómata Finito Probabilista que es solución de un problema se parece mucho a un Autómata Finito Determinista, bastará seguir con la vista las transiciones más oscuras. Esto facilitará al usuario la identificación de las transiciones importantes.

4.4.2. Resolver problema desde cadenas

Esta opción permite resolver el problema de otro modo. Consiste en la resolución de un único problema con una sola configuración. Ni el problema ni la solución serán enviados a la base de datos del servidor. La utilidad de esta opción es resolver problemas sueltos desde un ordenador.

Al hacer clic sobre esta opción, se abrirá una ventana donde aparecen los parámetros de configuración: estados, población, muestras, cruzador, mutador, calculador de bondad y opción de resolver.

Después de introducir la configuración, se deberá pulsar sobre modificar cadenas. Al hacer clic, se abrirá otra ventana donde aparecerán dos listas para introducir las palabras que deberán ser aceptadas o rechazadas.

Tras introducir las cadenas, se pulsará sobre resolver y aparecerá el autómata solución. Debajo aparecerán las estadísticas de reconocimiento y de parecido con autómata finito determinista.

4.5. Menú Estadísticas

Desde este menú se pueden consultar las estadísticas que muestran los problemas de la base de datos.

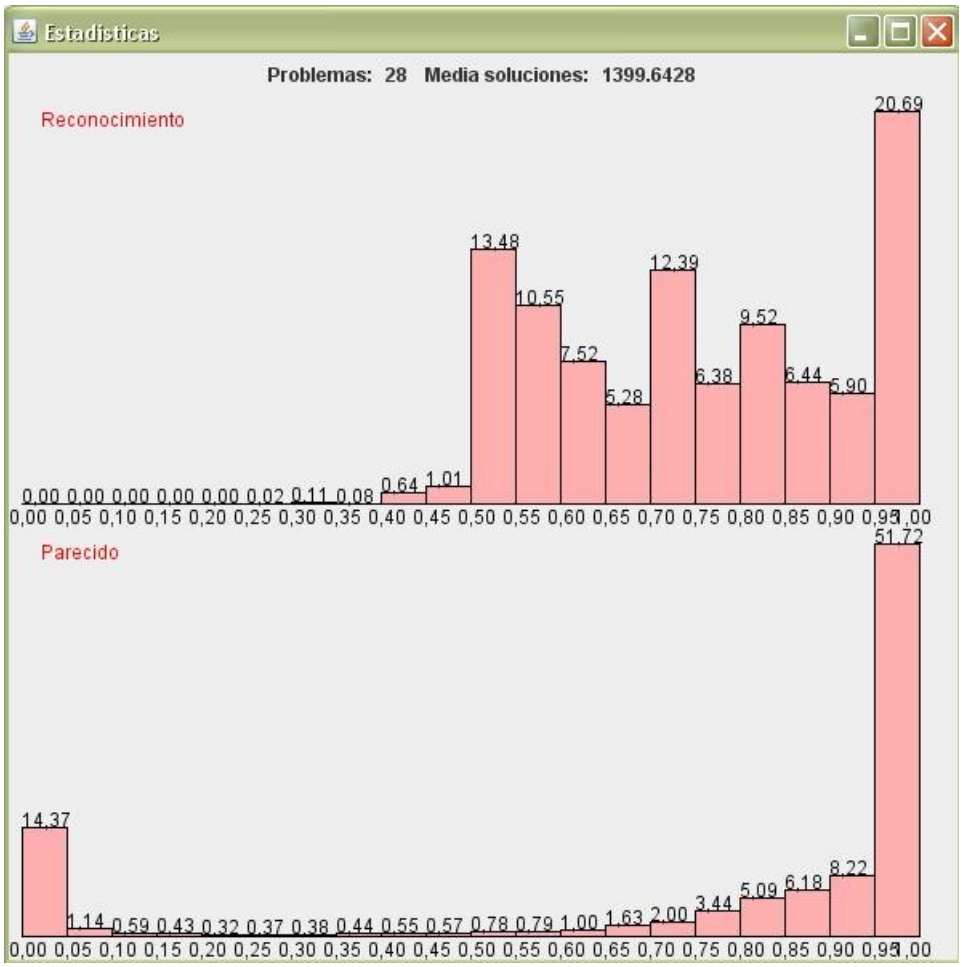
4.5.1. Ver estadísticas básicas

Esta opción muestra mediante dos gráficos de barras las estadísticas recogidas. Además, aparece el número de problemas existentes en la base de datos, y la media de soluciones por problema. Por tanto, el número total de soluciones obtenidas es, aproximadamente, el producto del número de problemas por la media de soluciones.

En el primer diagrama de barras, se recogen los porcentajes de reconocimiento que se han obtenido de las soluciones. Es decir, aparece el porcentaje de soluciones que encajan en un determinado porcentaje de reconocimiento de las cadenas.

En el segundo diagrama de barras, se recogen los porcentajes de parecido con los autómatas finitos deterministas de las soluciones. Es decir, el porcentaje de soluciones de la base de datos que encajan

en un determinado porcentaje de parecido con un autómata finito.



4.5.2. Ver estadísticas avanzadas

Al hacer clic en estadísticas avanzadas, se podrán ver otro tipo de estadísticas. Se podrán filtrar datos, de tal forma que aparezcan sólo determinados problemas con determinadas características.

Aparecerá una ventana que muestra la lista de problemas de la base de datos. Se podrá ir haciendo clic en cada problema que se desee añadir y pulsando en Añadir problema. Cuando se haya terminado, se deberá pulsar en Usar seleccionados.

Si se ha seleccionado sólo un problema, saldrá un panel que permitirá hacer el mismo tipo de selección, pero con las configuraciones de ese problema asociado. Si se ha seleccionado más de un problema, no aparecerá este panel de selección de configuraciones, ya que las configuraciones de los dos problemas pueden ser muy distintas.

A continuación, saldrá un panel donde se pueden filtrar las soluciones, mediante la selección de los parámetros elegidos. Se podrán elegir, los elementos concretos de los parámetros que se desee:

cruzadores, mutadores, poblaciones... Y sólo se mostrarán en dos gráficas, como las de las Estadísticas básicas, las soluciones que usaban en su configuración esos parámetros concretos.